2-1-2006

Path Planning Algorithms under the Link-Distance Metric

David Phillip Wagner
Dartmouth College

Path Planning Algorithms under the Link-Distance Metric

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

in

Computer Science

by

David Phillip Wagner

DARTMOUTH COLLEGE

Hanover, New Hampshire

February, 2006

Examining Committee:

(co-chair) Robert L. (Scot) Drysdale III

(co-chair) Clifford Stein

Amit Chakrabarti

Joseph S. B. Mitchell

Charles K. Barlowe
Dean of Graduate Studies

Abstract of the Dissertation

Path Planning Algorithms under the Link-Distance Metric

by

David Phillip Wagner

Doctor of Philosophy in Computer Science

Dartmouth College, Hanover, NH

February 2006

Professor Robert Scot Drysdale, Co-Chair

Professor Clifford Stein, Co-Chair

The Traveling Salesman Problem and the Shortest Path Problem are famous problems in computer science which have been well studied when the objective is measured using the Euclidean distance. Here we examine these geometric problems under a different set of optimization criteria. Rather than considering the total distance traversed by a path, this thesis looks at reducing the number of times a turn is made along that path, or equivalently, at reducing the number of straight lines in the path.

Minimizing this objective value, known as the link-distance, is useful in situations where continuing in a given direction is cheap, while turning is a relatively expensive operation. Applications exist in VLSI, robotics, wireless communications, space travel, and other fields where it is desirable to reduce the number of turns.

This thesis examines rectilinear and non-rectilinear variants of the Traveling Salesman Problem under this metric. The objective of these problems is to find a path visiting a set of points which has the smallest number of bends. A 2-approximation algorithm is given for the rectilinear problem, while for the non-rectilinear problem, an $O(\log n)$-approximation algorithm is given. The latter problem is also shown to be NP-Complete.

Next, the Rectilinear Minimum Link-Distance Problem, also known as the Minimum Bends Path Problem, is considered. Here the objective is to find a rectilinear path between two points among rectilinear obstacles which has the minimum number of bends, while avoiding passing through any of the obstacles. The problem has been well studied in two dimensions, but is relatively unexplored in higher dimensions. A main result of this thesis is an $O(n^{5/2} \log n)$ time algorithm solving this problem in three dimensions. Previously known algorithms have had worst-case running times of $\Omega(n^3)$.

This algorithm requires a data structure that supports efficient operations on pointsets within rectangular regions of the Euclidean plane. We design a new data structure, which is a variation on the segment tree, in order to support these operations.

Finally, an implementation of the data structure and of the algorithm solving the Minimum Link-Distance Problem demonstrates their experimental running times and ease of implementation.

Biographical Sketch

David Phillip Wagner was born in Aberdeen, Maryland on February 5, 1971. At the age of three his family moved to Bloomfield, Connecticut where he lived until the age of eighteen. In August 1989 he left home to attend Colgate University in Hamilton, NY.

At Colgate he earned a Bachelor of Arts degree with a dual concentration in Computer Science and Mathematics, working under advisors Phil Mulry in Computer Science and Tom Tucker in Mathematics, in May 1993. From there he went on to pursue a Master of Science degree in Computer Science at the State University of New York at Stony Brook, working under advisor Steven Skiena, and graduating in May 1995.

After finishing his Master's degree he worked for three years in New York City. The first six months of these were spent working at Wanderlust Interactive in SoHo, and the remainder was spent working downtown in the financial district for JYACC/Prolifics.

In September 1998 he returned to the academia to pursue a Doctorate in Computer Science at Dartmouth College, under co-advisors Clifford Stein and Robert Scot Drysdale.

It is perhaps interesting to note that although Colgate calls itself a liberal arts college, it retains the title of "University" due to a small graduate program which awards about ten Master's degrees each year. And, although Dartmouth has an active graduate program including schools of business, medicine, engineering, and dozens of fields in the arts and sciences, it retains the title of "College" due to the outcome of an 1818 Supreme Court case where a state run Dartmouth University was prevented from replacing Dartmouth College.

Acknowledgments

I would like to extend my heartfelt thanks to my thesis committee for their advice and the time they have spent considering this dissertation, and for the many fruitful discussions they have provided. I would especially like to thank my co-advisor Robert Scot Drysdale whose fastidious attention to the details of my work have proven invaluable, especially in Chapters 4 and 5, and my co-advisor Clifford Stein who inspired me to pursue this line of research in a moment of creativity and who advised me throughout the writing of Chapter 3. I must also thank Neal Young, who guided me through the very earliest incarnations of this research, especially that appearing in Sections 3.3 and 3.4.1.

Along the way, many talented researchers have given their time and advice to further the results in this thesis. These include Michel X. Goemans (Section 3.4.1), Takeshi Tokuyama (Section 4.4.2), David R. Karger (Chapter 2), Jon L. Bentley (Section 7.2.1), Günter Rote (Section 7.2.1), Tatsuo Ohtsuki (Section 1.1), Joseph S. B. Mitchell (Sections 1.1, 3.6.5, and 7.2.4), Valentin Polishchuk (Section 1.1), and Robert Fitch (Chapters 2 and 4). I am grateful for their advice which illuminated many deficiencies, and allowed many further results within this work.

I am deeply indebted to numerous faculty who have encouraged, enlightened, and inspired me through the years, and who have taught me to appreciate a lifelong pursuit of learning. I would especially thank Prasad Jayanti, Javed Aslam, Amit Chakrabarti, and Kenneth Bogart from Dartmouth, Steven Skiena and Joseph Mitchell from SUNY Stony

Brook, and Laura Sanchis, Phil Mulry, Tom Tucker, and Dan Saracino from Colgate. It would not be an exaggeration to say that every one of these people has substantially changed my life for the better.

I am very grateful to the NSF Summer Institute Program, for their sponsorship of research summers at IBM Tokyo Research Laboratory in Yamato, Japan, and at Korea Advanced Institute of Science and Technology (KAIST) in Taejon, Korea. The memories I have from these trips will undoubtedly last a lifetime. I consider myself incredibly fortunate to have met many researchers abroad who were kind enough to invite me into their laboratories in order to broaden my understanding of how research is done, including Takao Nishizeki from Tohoku University, Takeshi Tokuyama from Tohoku University, Satoru Iwata from Tokyo University, Tetsuo Asano from JAIST, Kyung-Yong Chwa from KAIST, Kyunsoo Park from Seoul National Universtiy, Mordecai Golin from HKUST, Tak-Wah Lam from Hong Kong Universtiy, and Der-Tsai Lee from Academia Sincia.

Additionally, the faculty of the Department of Japanese at Dartmouth and at Stony Brook have been very gracious in repeatedly permitting me to audit their classes. I am grateful especially to Mayumi Ishida, Ikuko Watanabe, Dennis Washburn, and James Dorsey. Without their help I surely would have forgotten a fascinating and beautiful language.

Many clubs and organizations I have participated in while at Dartmouth have made my time here rich and enjoyable. I am fortunate to have been involved with the Handel Society, the Dartmouth Japan Society, the Dartmouth Argentine Tango Society, Pound Bang Slash Intramural Hockey Team, Korean Miniversity, UFX Ultimate Frisbee, MEaD, and the Dartmouth Bridge Club, among others. Thanks to Bhavnesh at the India Queen, Ken and the uvScene, the Tabard, the Hop, and the Lebanon Opera House for providing many ways to warm up those cold New Hampshire nights.

My friends at Dartmouth have undoubtedly made this a more memorable place. Ed, Chilann, Marina, You-Shan, Luma, my two years in North Park were some of the best of

my life. LeeAnn, Ming, Kazuhiro, Zhengyi, this department has been a richer place thanks to you. Guanling, Tony, Clint, Mark, Lea, Song, I'm fortunate to have had such wonderful officemates. Ji Yeon, James, Kim, thanks for many helpful language lessons. George, Sue Mei, Helen, and all of DJS, I hope we can meet at another karaoke bar someday. Wendy, Ashley, Yulie, I'll miss seeing you at those medieval feasts. Liya, Katie, Brian, Barry, Karen, Daisy, good luck with the Argentine Tango club.

I especially want to thank some friends I've made outside of Dartmouth. Robyn, Lauren, and everyone at the Center for Cartoon Studies in White River Junction, I hope you and your fledgling school will be a great success. Wendy, Devin, Helen and the rest of the Summer Institute, how can I ever forget our summer in Tsukuba and Tokyo. Sangmin, Hyunwoo, all of KAIST's theory of computing lab, and especially Hyoung-Shim, Korea will always be a special place for me.

Finally, I would like to thank my family, my Mom, my Dad, my grandparents, my sister Alicia, and my brother-in-law Billy for their ongoing love and encouragement. Without their support this dissertation would not have been possible.

For Hyoung-Shim Choi,

my Daramjui

('v')

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Numerous problems in Computational Geometry involve the objective of trying to minimize the total cost of a path, a spanning tree, a set of paths, or a similar geometric structure, while achieving some additional objective, such as connecting a pair of points or a set of points, and possibly avoiding a set of obstacles. Among the most famous of these are the *Euclidean Traveling Salesman Problem*, where the objective is to visit a set of points in the plane via a tour having the minimum distance, and the *Euclidean Shortest Path Problem* where the objective is to find a minimum distance path between two points in the plane which avoids a set of obstacles. Surveys of path planning problems in computational geometry have given substantial consideration to these two problems [120, 122].

In this thesis we investigate these problems under a different set of optimization criteria, that of minimizing the number of bends. In most cases, the nature of problems under this metric are quite different from their distance-minimizing counterparts. For example, it is often beneficial for a straight segment of a path to extend well beyond the limits of any obstacle or point before turning, so that it can meet up with another segment without inducing unnecessary bends.

Minimizing the number of bends in a path is equivalent to minimizing the number of

straight line segments comprising the path, since a path between two points or an open tour comprised of a series of straight line segments has exactly one more line segment than it has bends, and a closed tour which returns to the starting location has the same number of bends as line segments. Thus, these problems can be discussed in terms of minimizing the number of straight line segments, or *links*, in the path or alternately in terms of minimizing the number of bends, as the two objectives are equivalent.

The number of straight line segments in a path is often referred to as the *link-distance* or the *link length* of the path or tour, and the number of bends is referred to as the *bend distance* or *bend number*. A path having a minimum link-distance (or bend distance) is called a *minimum link path*. A path which passes through all of a set of points is often called a *covering tour* or a *spanning path*. A covering tour having a minimum link-distance is called a *minimum link tour*.

The objective of minimizing the link-distance was studied as early as 1968 by Mikami and Tabuchi in the context of VLSI design [117]. Their $O(n^2)$ algorithm finds a rectilinear minimum link-distance path between two query points among rectilinear obstacles in two dimensions. Their method uses a *Breadth First Search (BFS)* strategy, exploring the cells of the grid, G, defined by the set of all lines containing obstacle edges. See Section 4.2.1 for a discussion of Breath First Search approaches to this problem.

In the 1980's Sato, Sakanaka, and Ohtsuki began investigating strategies to improve on the running time of the Mikami-Tabuchi algorithm. Their idea was to eliminate the use of G and instead use a rectangular partition of the plane. The resulting "Gridless" algorithm was outlined in 1986 technical report, written in Japanese, and the algorithm runs in $O(n \log n)$ time [146]. This appeared in an English publication in 1987 [147]. Yang, Lee, and Wong would later give a more detailed description of this approach, expanding it to solve many related problems [94]. A similar method was outlined in 1991 in an independent result of Das and Narasimhan [38].

The non-rectilinear version of this problem was first examined by Suri in 1986 [151], although ElGindy laid the groundwork for the single-source version of the non-rectilinear problem in his 1985 Ph.D. thesis [50]. Suri gave an $O(n)$ algorithm to find a minimum link-distance path between two query points inside a triangulated simple polygon, and ElGindy's method runs requires $O(n \log n)$ preprocessing in order to support $O(\log n)$ queries. Suri eventually improved the preprocessing time of ElGindy's approach to $O(n)$ plus the time to triangulate the polygon[153]. Shortly thereafter, Chazelle described how to triangulate a simple polygon in linear time, giving both of Suri's approaches overall linear running times [31].

Since then, many publications on the subject have appeared. The Ph.D. theses of Suri and of Piatko each discuss several varieties of link-distance problems [152, 139]. More recently, a survey of link-distance results has been been included in a Computational Geometry textbook [112]. Section 1.1 below gives an overview of many important results on link-distance problem.

Here, this thesis considers in particular the *Minimum Link Traveling Salesman Problem*, also known as the *Minimum Link Tour Problem*, or the *Minimum Bends Traveling Salesman Problem*, where the objective is to find a tour consisting of the smallest number of straight line segments such that each of a given set of points falls along some line segment of the tour.

Additionally, the *Minimum Link Path Problem*, also known as the *Minimum Bends Path Problem*, where the objective is to find a path between two points consisting the smallest number of straight line segments which do not intersect a given set of obstacles, is considered here.

The value n will be used in this thesis to denote the size of the problem under consideration. In the case of *tour problems* where the objective is to visit a set of points, this value denotes the number of points to be visited. In contrast, this value denotes the number of

corners among all obstacles in the case of *path problems*, where the objective is to connect two points with a path which avoids the obstacle region or set of regions. Another class of tour problem is that where a region is covered instead of a set of points. In this case, the size of the problem is the number of corners in the region being covered.

Problems of this nature have numerous applications in a wide variety of areas. They have proven particularly useful in VLSI, where a bend often implies a connection between two wires, as well as in robotics, in situations where a robot can move forward with ease but where turning is a difficult prospect. Chapter 2 describes a number of these applications in more detail.

1.1 Previous and Related Work

This thesis draws upon a rich body of existing research in the field of link-distance problems. In this section many of the most important of these results are revisited. This section is divided into four parts: the first lists results concerning tour problems, including those which cover a region; the second lists results about path problems; the third lists bicriteria path problems, where there are two objectives being optimized simultaneously; and the last mentions some alternate computing models which have been studied.

Tours problems, where the objective is to find a path such that each of a set of points lies along some line segment of the tour, will typically have no obstacles. Path problems, where the objective is to connect two points with a path, will typically have obstacles which obstruct the path. Tour problems which cover an area or space, such as watchman tours and lawn-mowing tours, will have a target region to be covered, and this region may or may not restrict the tour.

Both rectilinear and non-rectilinear variations of problems are considered here. When rectilinear versions are discussed, all line segments of the target path are required to be par-

allel to some axis, and distance is measured using the L_1 *norm*. Otherwise, the Euclidean distance, also known as the *geodesic* in order to distinguish it from the non-Euclidean length of a path through a graph with edge weights, or as the L_2 *norm*, is used.

If there are obstacles in the problem under consideration, and if the problem is non-rectilinear then the obstacles may either be a simple polygon, or an arbitrary set of obstacles, sometimes called a polygon with holes. If the problem is rectilinear, then within the context of this thesis, any obstacles are also required have axis-parallel edges, and usually may have holes. These requirements on obstacles usually, although not always, exist when such problems are discussed in the literature.

In many problems, especially tour problems, finding an optimal solution is NP-Hard. Thus, the algorithms which attempt to solve these problems are approximation algorithms. A ρ-approximation algorithm is one which find a solution whose measure is provably within a factor of ρ of the optimal solution.

1.1.1 Tour Problems

- **[Minimum Total Turn Tour / Angular-Metric Traveling Salesman Problem]** In the *Minimum Total Turn Tour Problem*, also known as the *Angular-Metric Traveling Salesman Problem*, the objective is to find a tour π consisting of a series of straight line segments $\{l_1, l_2, \ldots, l_m\}$ such that each of a set of points $P = \{p_1, p_2, \ldots, p_n\}$ falls along some line segment $l_x \in \pi$, where the sum of all angles of all turns along the path is minimized. This sum is referred to by Mitchell, Piatko, and Arkin as the *Total Turn* [123, 139]. The problem is later discussed by Aggarwal, Coppersmith, Khanna, Motwani, and Schieber who give a polynomial time $O(\log n)$-approximation algorithm. They also show that the problem is NP-Complete [2].

- **[Minimum Link Traveling Salesman Problem / Minimum Link Tour]** The *Mini-*

mum Link Traveling Salesman Problem, also known as the *Minimum Link Tour Problem*, is similar to the Angular-Metric Traveling Salesman Problem, but uses the total number of bends as the cost function, instead of the total angle of all turns. Rectilinear and non-rectilinear variants of the problem are discussed in [150] and Chapter 3 of this thesis. An $O(\log n)$-approximation algorithm is given for the non-rectilinear version and a 2-approximation algorithm is given for the rectilinear version. For the case when the points to be visited are in general position, meaning that no two points share an x-coordinate or a y-coordinate, and the tour is rectilinear, an algorithm which finds a tour which has at most two additional bends beyond the optimal, is given. This variant is known as the *Non-Collinear Rectilinear Minimum Link Traveling Salesman Problem*. The non-rectilinear problem has been shown to be NP-Complete by Arkin, Mitchell, and Piatko [11], and also independently in Chapter 3 of this thesis. Note that the rectilinear version of this problem and the rectilinear version of the Angular-Metric Traveling Salesman Problem are equivalent.

- **[Minimum Link Tour of a Grid]** The *Minimum Link Tour of a Grid Problem* is identical to the Rectilinear Minimum Link Traveling Salesman Problem, except that the points of P are confined to a subset of the vertices of a d-dimensional grid. A special case of this problem, where P is the complete set of all vertices of a $n \times n$ grid without holes, is considered by Kranakis, Krizanc, and Meertens [91]. Despite the simplicity of the problem in two dimensions, it becomes non-trivial to optimize in higher dimensions. They describe a trivial algorithm solving the problem in two dimensions, and give an approximate and possibly optimal algorithm for three dimensions. Their method finds a tour of an $n \times n \times n$ grid having $\frac{3}{2}n^2 + n - 1$ links, which yields an immediate approximation factor of $\frac{3}{2} + O(1/n)$. A lower bound of $\frac{7}{6}n^2$ was shown for this problem by Collins and Moret in 1998 [37], and this was

improved to $\frac{4}{3}n^2 - O(n)$ by Collins in 2004 [36], yielding an approximation ratio of $\frac{9}{8} + O(1/n)$. Both [91] and [37] discuss expanding their methods into higher dimensions. Optimal algorithms for some additional simple two dimensional point configurations, such as pairs of rectangular grids, and the grid falling inside a triangle are discussed by Collins [36]. Approximation algorithms for the more general problem of covering a two dimensional grid of points with holes are a consequence of results given by Arkin, Bender, Demaine, Fekete, Mitchell, and Sethia [6].

- **[Minimum Link Watchman Tour / Visibility Path]** The *Watchman Tour Problem*, sometimes called the *Visibility Path Problem*, attempts to find a tour of a polygonal region, R, possibly with holes, such that every point within the interior of R is visible from some point along the tour. That is, there exists a straight line segment which does not intersect the exterior of R, between every point on the interior of R and some point along the tour. The *Minimum Link Watchman Tour Problem* attempts to find a watchman tour consisting of the smallest number of straight line segments. This problem is related to the Minimum Link Traveling Salesman Problem in the sense that both problems look for tours which cover a set of points.

  Alsuwaiyel and Lee show that this problem is NP-Complete if R is a simple polygon without holes [4]. They give further results for this case [5], including a 4-approximation algorithm with running time $O(n^2)$, and a 3.5-approximation algorithm with running time $O(n^3)$. Arkin, Mitchell, and Piatko use the similarity to the Minimum Link Traveling Salesman Problem to show that this problem is NP-Complete, even if R is restricted to being a convex polygon with convex holes. They additionally give an $O(\log n)$-approximation algorithm for the case when R can be any polygon with holes [11]. Note that Alsuwaiyel and Lee's NP-Completeness result trivially reduces to the case when R can be non-convex with holes, although their

approximation algorithms do not apply.

- **[Minimum Link Milling / Lawn Mowing / Ice Rink Problem]** In the *Milling Problem* one is is given a Euclidean region R to cover with some shape C, known as a *cutter*. Typically C is a square or circle. The objective is to move C about in a tour of R such that every point within R falls within the boundary of C at some point during the tour. The movement of C is restricted so that it may not leave R. This problem was first studied as a distance minimization problem [9]. The link-distance version is considered by Arkin, Bender, Demaine, Fekete, Mitchell, and Sethia in [6] and these results are expanded in [7]. They give a number of approximation algorithms for different variations of this problem (See Table 1.1 for a list of approximation ratios). The *Lawn Mowing Problem* is identical to the Milling Problem except that C may travel beyond the boundaries of R. These problems have also been referred to in a more general sense as the *Ice Rink Problem* [125]. The rectilinear problem is closely related to covering tours of grids. If all coordinates are integral multiples of the size of a square cutter, then the problem is identical to the Minimum Link Tour of a Grid (with holes), although if the grid points are represented explicitly, then the size of the problem representation may differ substantially.

1.1.2 Path Problems

- **[Minimum Link Path]** The *Minimum Link Path* problem attempts to find a path, π, consisting of a series of straight line segments, $\{l_1, l_2, \ldots, l_m\}$, between two points s and t such that π avoids intersecting any of a set of obstacles R, and the number of straight line segments, m, in the path is minimized. The problem has been well studied in two dimensions, mostly owing to its applications in the area of VLSI, and also for this reason much attention has been paid to rectilinear versions of the

problem.

It was shown by Suri in 1986 that the minimum link-distance path between two points in a simple triangulated polygon can be found in $O(n)$ time. This was one of the first results among non-rectilinear link-distance problems [151]. Chazelle later showed how to triangulate a simple polygon in linear time [31], improving Suri's method to overall linear running time. Mitchell, Rote, and Woeginger have shown how to find the minimum link-distance path in a polygon with holes in nearly quadratic $O(n^2\alpha(n)\log^2 n)$ time [124].

Rectilinear versions of the problem appeared as early as 1968, when Mikami and Tabuchi gave a breath first search style algorithm finding a rectilinear minimum link path among rectilinear obstacles in $O(n^2)$ time [117]. Sato, Sakanaka, and Ohtsuki improved on this in 1986, showing how to find a path in $O(n \log n)$ time [146]. This was independently studied by Das and Narasimhan, and in 1991 they also gave an $O(n \log n)$ time algorithm finding a rectilinear minimum link-distance path among rectilinear obstacles [38].

The rectilinear problem was studied in three dimensions by Fitch, Butler, and Rus, who expanded the Sato-Sakanaka-Ohtsuki algorithm into three dimensions. They gave an algorithm which runs in $O(n^3)$ time in the worst case, but which in many situations runs in $O(n^2 \log n)$ [54]. This result is improved in Chapter 4 of this thesis, with an algorithm which runs in $O(n^{5/2} \log n)$ in the worst case, but which often provably runs in $O(n^2 \log n)$. This result was published in 2005 [41].

- **[k-Reachability / k-Visibility]** Given a polygon the *k-Reachability Problem*, also known as the *k-Visibility Problem*, asks for the region of that problem which is reachable within k bends of a starting point.

  ElGindy considered the non-rectilinear problem in his 1985 Ph.D. thesis. He gave an

$O(kn \log n)$ time algorithm to compute the k-reachable reachable region of a simple polygon. He also described how to improve this time to $O(n \log n)$ [50]. Asano looked at the rectilinear version of this problem that same year. He showed how to find the k-reachable region of a simple rectilinear polygon in $O(kn)$ time [14].

- **[Link Center / Link Diameter]** The *link center* of a polygonal region, R, is the subregion of R containing the set of points for whom the maximum among the link-distance to any point in R is minimized. Note that the link center does not necessarily lie within R. A related value is the *link diameter* which is the maximum link-distance between any two points on the boundary of R.

  In his 1987 Ph.D. thesis Suri described how to compute the link diameter of a simple polygon in $O(n \log n)$ time [152]. In 1988 Lenhart, Pollack, Sack, Seidel, Sharir, Suri, Toussaint, Whitesides, and Yap showed how to find the link center of a simple polygon in $O(n^2)$ time [98]. The link center result was improved by Ke in 1989 [82] and independently by Djidjev, Lingas, and Sack in 1992 [40] who each describe how to find the link center and link diameter of a simple polygon in $O(n \log n)$ time.

  Rectilinear results appeared in 1991, when Nilsson and Schuierer showed how to find the link diameter of a simple rectilinear polygon in linear time [126]. They showed later how to find the link center of a simple rectilinear polygon in linear time [127].

  In 1996 Bose and Toussaint described how to find the *constrained link center* of a simple polygon in $O(n \log n)$ time, where the constrained link center is the subset of points along the boundary of R for whom the maximum link-distance to any point within R is minimized [25].

Preprocessing

In many cases the running time of Link-Distance problems can be improved with some preprocessing. Some of the most significant of those results are mentioned here.

- **[Minimum Link Path / Link-Distance Query]** If one only wants to find the link-distance between two query points, and not the path itself, then the problem is called the *Link-Distance Query Problem*. With some preprocessing, link-distance queries can frequently be answered in logarithmic time, and the path itself can then be recovered in time proportional to the link length of the path.

  With $O(n^3)$ preprocessing time one can perform link-distance queries between two arbitrary points s and t in a simple polygon in $O(\log n)$ time, in a result of Arkin, Mitchell, and Suri. The path itself can then be recovered in $O(\log n + k)$ time, where k is the link-distance of the path [12].

  The rectilinear version of the problem was addressed by de Berg in 1991. He shows that, in a simple rectangular polygon, it is possible to perform $O(\log n)$ time queries between two arbitrary points, s and t, and to recover the path in $O(\log n+k)$ time with $O(n \log n)$ preprocessing. They also show that this path has the minimum geodesic, making it a smallest path (see below) [21].

- **[Single-Source Minimum Link Path / Single-Source Link-Distance Query]** The Single-Source Minimum Link Path Problem attempts to improve on the running time of the Minimum Link Path Problem by restricting all paths to a common starting point and preprocessing the instance for future queries in order to take advantage of this restriction.

  In his 1985 Ph.D. thesis, ElGindy describes how to compute the *weak visibility polygon* from an edge, e, of a simple polygon R. This is the union of all points within

R which are visible from any point of e, where visibility implies the existence of a straight line segment which does not exit R, between the points and e. In Section 3.6 of the thesis, he explains that by repeatedly finding weak visibility polygon, one can compute the k-*reachable* subregion of R, from a source point, s, in $O(n \log n)$ time. This is the subregion of R which is reachable by a path from s, entirely interior to R, and having at most k bends [50].

The k-reachable regions can then be augmented with a point location data structure, such as that described by Sarnak and Tarjan [145]. The resulting structure supports non-rectilinear single source minimum link-distance queries in the polygon with $O(\log n)$ query times and $O(\log n + k)$ path recovery times.

This running time was obtained independently by Reif and Storer in 1987 [141].

The preprocessing time on this problem was improved by Suri in 1990, who used a *triangulation* in place of ElGindy's map of k-visibility regions [153]. He achieved a running time of $O(n)$ plus the time to triangulate the polygon (which at the time took $O(n \log \log n)$ [156]). Shortly afterwards, Chazelle showed how to triangulate a simple polygon in $O(n)$ time [31], improving Suri's method to $O(n)$ preprocessing time overall.

Das and Narasimhan consider the single-source problem when dealing with rectilinear obstacles. They show that with $O(n \log n)$ preprocessing, the optimal rectilinear path to a query point can be found in $O(\log n)$ time, and the path recovered in $O(\log n + k)$ time [38].

1.1.3 Bicriteria Path Problems

Bicriteria Problems are a class of problems in which more than one objective is considered. In her 1993 Ph.D. thesis, Piatko discusses many results for a number of bicriteria path

problems [139]. Many bicriteria results have also been published by Yang, Lee, and Wong [160, 94, 161, 95, 162].

Perhaps the most important among bicriteria problems are those which consider some combination of the link-distance of the path and the total Euclidean distance (also known as the geodesic or the L_2 norm). This is a very practical consideration, since in many applications turning and moving forward both restrict or contribute to the total cost of operations.

A number of bicriteria results are listed here.

- **[Smallest Path]** A *Smallest Path* is one which simultaneously minimizes both the Euclidean distance and the link-distance of the path. Note that there may not always exist a smallest path depending on the problem type and the instance.

  Arkin, Mitchell, and Suri showed that in a simple polygon, there always exists a path which simultaneously is an approximation of both the link-distance and the geodesic. Given any minimum link path, they show how convert it into a path for which the link-distance is at most double that of a minimum link path, and for which the Euclidean distance is at most $\sqrt{2}$ times the length of a minimum geodesic path, in time proportional to the link-distance of the path [12].

  In 1991, de Berg showed that a smallest rectilinear path always exists between two points in a simple rectilinear polygon, giving an $O(n \log n)$ algorithm finding such a path. Their method also yields a solution to the single-source version in $O(n \log n)$ preprocessing and $O(\log n)$ query time [21]. McDonald and Peters improve on the two point query problem in 1992, giving an algorithm that reports a smallest path in $O(n)$ time [115].

  Yang, Lee, and Wong discuss finding two non-crossing smallest paths between two pairs of points inside a simple rectilinear polygon. Although such a pair may not

always exist, they show how to find it in $O(n)$ time if it does [162].

- **[Minimum Cost Path / Combined Metric]** The *Minimum Cost Path Problem* assigns the cost of a path to be the total Euclidean length of the path, plus some non-decreasing function, f, of the number of bends, b, in the path. By setting $f(b) = 0$ or $f(b) = Cb$, where C is a sufficiently large number, this problem becomes the Euclidean Shortest Path Problem or the Minimum Link Path Problem respectively.

  Yang, Lee, and Wong give an algorithm for the rectilinear version of the problem that finds the minimum cost path in $O(nt + n \log n)$ time. Here t is the number of extreme edges among all obstacles, where an extreme edge is defined to be one with two convex corners [160].

  When f is a linear function, this is called the *Combined Metric Problem*. In one of the first link-distance results in arbitrary dimensions, de Berg, van Kreveld, Nilsson, and Overmars showed in 1992 how to compute the optimal combined metric path of the rectilinear problem in d dimensions in $O(n^d \log n)$ time [22].

- **[Shortest Minimum Link Path / Shortest Minimum Bend Path]** In the *Shortest Minimum Link Path Problem* (also called the *Shortest Minimum Bend Path*) the objective is to find the path having the shortest Euclidean distance among all paths which already have the minimum link-distance. Note, this does not necessarily yield a path with the shortest Euclidean distance. Lee, Yang, and Wong give an algorithm solving the rectilinear version of this problem in $O(n \log n)$ time [94].

- **[Shortest k-Link Path / Bounded-Bend Shortest Path]** The *Shortest k-Link Path Problem*, also known as the *Bounded-Bend Shortest Path*, has the objective of finding the path with the shortest Euclidean distance among all paths which already have a link-distance of at most k. This is more than just the decision version of the Shortest

Minimum Bend Path Problem, as one can often find a shorter path than the shortest minimum bends path when k is not optimized. Note that a shortest k-link path will not exist if k is too small.

Mitchell, Piatko, and Arkin give a *Polynomial Time Approximation Scheme (PTAS)* finding a path through a simple polygon, R, where the running time varies in a trade-off with the desired approximation factor, which can be arbitrarily close to one. This algorithm runs in $O(n^3k^3 \log{(Nk/\epsilon^{1/k})})$ time, where N is the largest integer coordinate among the vertices of R, and $(1+\epsilon)$ is the approximation factor [123].

They also consider the case when R is a polygon with holes, giving an algorithm that finds a path that has the same distance as the optimal shortest k-link path, and has at most $2k$ links. This runs in $O(kE^2)$ time, where E is the number of edges in the visibility graph of R.

The rectilinear version of the problem was considered in 1992 by Yang, Lee, and Wong [160]. They give a graph theoretic algorithm solving the problem $O(k(nt + n \log n))$ time, where t is the number of extreme edges among the obstacles. The problem was recently revisited in 2005 by Polishchuk and Mitchell. They describe an algorithm that runs in $O(kn \log^2 n)$ time [140].

- **[Minimum Total Turn k-Link Path]** In the *Minimum Total Turn k-Link Path Problem* the objective is to find a path comprised of at most k links where the sum of the angles of all turns in the path is minimized. Piatko describes how to find this path in a polygonal region with holes, R, in $O(E^3 n\alpha(n) \log^2 n)$ time, where E is the number of edges in the visibility graph of R, and $\alpha(n)$ is the inverse of Ackermann's function. The case when R is a simple polygon is also considered, and a linear time algorithm is given [139].

- **[Minimum Link Shortest Path / Minimum Bend Shortest Path]** In the *Minimum Link Shortest Path Problem*, also called the *Minimum Bend Shortest Path Problem*, the objective is to find a path with the minimum link-distance among those which already have the shortest Euclidean distance. This does not necessarily yield a path with the minimum link-distance. Yang, Lee, and Wong describe an $O(nt + n \log n)$ time algorithm solving this where t is the number of extreme edges among the obstacles. Note this running time could be as large as $O(n^2)$ [160]. They later give an $O(n \log^{3/2} n)$ time algorithm [161].

- **[Minimum Total Turn Shortest Path]** The problem of finding a tour which minimizes primarily the Euclidean distance of the tour and secondarily the total angle turned during the tour is called the *Minimum Total Turn Shortest Path Problem*. Piatko shows the problem to be NP-Complete, but only weakly so, if the obstacle set R can have holes. She also gives a pseudo-polynomial time algorithm [139]. This algorithm runs in $O(En^2N^2)$ time, where E is the number of edges in the visibility graph of R and the vertices of the obstacle scene lie within an $N \times N$ grid.

1.1.4 Alternate Computing Models

A number of parallel algorithms have been developed for these problems. Among the most notable such results are those of Chandru, Ghosh, Maheshwari, Rajan, and Saluja for the Minimum Link Path Problem in a simple polygon [29], and those of Lingas, Maheshwari, and Sack, concerning the Minimum Link Path Problem among rectilinear obstacles [103]. A summary of their results as well as some additional references to parallel link-distance algorithms appears in [112].

Additionally, Kahan and Snoeyink have discussed the bit complexity of some non-rectilinear link-distance problems [79].

Only sequential algorithms under the RAM model are discussed within this thesis, and so the interested reader is referred to the above mentioned papers for further details.

1.2 Our Contributions

The primary contributions of this thesis are to the Minimum Bends Traveling Salesman Problem, and to the Rectilinear Minimum Link Path Problem in Three Dimensions.

This chapter (Chapter 1) introduces the problems that will be considered and gives some background history of the work done on a number of various link-distance problems.

Chapter 2 lists a variety of applications for this class of problem. These include many that are instances of the problems discussed above in Section 1.1.

In Chapter 3 the Minimum Link Traveling Salesman Problem is studied. Several variations of this problem are considered, including rectilinear and non-rectilinear variants. Here it is shown that the non-rectilinear problem is NP-Complete, based on a reduction from Line Cover. This theorem was independently proven by Arkin, Mitchell, and Piatko [11]. Additionally, using known approximation algorithms for Set Cover, an $O(\log n)$ approximation algorithm is discussed here for this variant.

The Rectilinear Minimum Link Traveling Salesman Problem problem can also be approximated. In this case a 2-approximation, which is based on algorithms solving Bipartite Vertex Cover, is described. It remains open if this rectilinear variant is NP-Complete, or is polynomially solvable.

Next, it is discussed what happens if no two points are horizontally or vertically collinear in the rectilinear problem. In this case an algorithm that comes within two bends of the optimal solution is given. A central idea of this algorithm is the classification of points into a set of box points and a set of diagonal points, with each set exhibiting certain properties. This classification arises solely from the non-collinearity property, and it may prove useful

in contexts beyond the problems discussed in this thesis.

Chapter 4 looks at the Minimum Link Path Problem in three dimensions. This problem has been studied extensively in two dimensions (see Section 1.1 above), but is relatively unexplored in three dimensions and higher. Existing algorithms solving the two-dimensional problem require $\Omega(n \log n)$ time in the worst case [38, 146, 94].

Previous algorithms solving the three-dimensional problem required $\Omega(n^3)$ in the worst case [54, 22]. In this chapter, an algorithm which solves the three dimensional problem in $O(n^{5/2} \log n)$ time in the worst case is described. This is achieved primarily by using a binary space partition of the scene containing the obstacles, and by using the data structure described in Chapter 5.

In Chapter 5 a new kind of data structure, which is a variation on the two-dimensional segment tree, is introduced. This data structure has the ability to perform a variety of operations on rectangular regions of points in the plane. These include inserting and clearing rectangular regions of points in the plane in $O(n \log n)$ time, and inserting a series of rectangles which all have common endpoints in one of the dimensions, in $O(n \log n)$ time. Further, it is possible to query a rectangular region in $O(\log^2 n)$ time to see if it contains any data, and the data contained within a rectangular region can be projected onto an axis in $O(n \log n)$ time. These operations are used in the algorithm described in Chapter 4.

1.3 Summary and Tables

Tables 1.1, 1.2, and 1.3 summarize the problems discussed in the previous sections. Listed therein are the running times, approximation factors, and the source of the best known algorithm which solves each of the problems. In some cases multiple running times and approximation factors are given for the same problem, when trade-offs can be achieved between the two values.

Tour Problems

Problem	Rect. Path?	Dim.	Restrictions on Obstacles	Approximation Factor	Running Time	Source
Minimum Total Turn Tour	No	2	no obstacles	$O(\log n)$	polytime [a]	[2] [b]
Minimum Link Tour [c]	No	2		$O(\log(\min(z, c)))$	$O(n^2)$	* [d], [150] [e]
	Yes			2	$O(n^{3/2})$	
Non-Collinear Min. Link Tour [f]	Yes	2		$OPT + 2$	$O(n \log n)$	
Minimum Link Tour of a Complete Grid	Yes	2		1	$O(n)$	[91]
		3		$1.125 + O(1/n)$	polytime [a]	[91, 36]
Minimum Link Watchman Tour	No	2	simple polygon	4	$O(n^2)$	[5]
				3.5	$O(n^3)$	
			polygon with holes	$O(\log n)$	$O(n^3)$	[11]
Minimum Link Milling/ Minimum Link Mowing/ Minimum Link Tour of a Grid with Holes [i]	Yes	2	rectilinear	6.25	$O(N^{2.376} + n^3)$	[6, 7] [g]
			integral rectilinear [h]	12	$O(N)$	
				6	$O(N^{2.376})$	
				3.75	$O(N^{2.376} + n^3)$	

[a] The authors of [2] and [91] do not discuss the running time of their algorithm, other than to say that it is polynomial.
[b] The authors of [2] call this problem the Angular Metric TSP.
[c] Here z is the maximum number of collinear points and c is the size of a minimum line cover.
[d] A '*' indicates that this dissertation is the source of the algorithm.
[e] In [150] this problem is called the Minimum Bends TSP.
[f] Note that this algorithm has an additive approximation bound, not a multiplicative bound.
[g] Here N is the ratio between the cutter and the largest coordinate.
[h] In the integral rectilinear problem all coordinates are multiples of the cutter size.
[i] The integral rectilinear variation of Lawn Mowing is closely related to finding a tour of a grid, primarily differing in the input representation.

Table 1.1: A list of results in the area of link-distance tour problems and the running times, approximation factors, and the source of the best known algorithm solving the problems.

Path Problems

Problem	Rect. Path?	Dim.	Restrictions on Obstacles	Preprocessing Time	Query Time	Source
Minimum Link Path	No	2	simple polygon	$O(1)$	$O(n)$	[151, 31]
			polygon with holes		$O(n^2\alpha(n)\log^2 n)$	[124] [a]
	Yes		rectilinear		$O(n \log n)$	[146, 147, 38]
		3			$O(n^{5/2} \log n)$	* [b] , [41]
		d			$O(n^d \log n)$	[22]
	No	2	simple polygon	$O(n^3)$	$O(\log n + k)$	[12]
	Yes		simple rectilinear	$O(n \log n)$	$O(\log n + k)$	[21]
Link-Distance Query	No	2	simple polygon	$O(n^3)$	$O(\log n)$	[12]
	Yes		simple rectilinear	$O(n \log n)$	$O(\log n)$	[21]
Single-Source Minimum Link Path	No	2	simple polygon	$O(n)$	$O(\log n + k)$	[153, 31]
	Yes		rectilinear	$O(n \log n)$	$O(\log n + k)$	[38]
		d		$O(n^d \log n)$	$O(\log^{d-1} n + k)$	[22]
Single-Source Link-Distance Query	No	2	simple polygon	$O(n)$	$O(\log n)$	[153, 31]
	Yes		rectilinear	$O(n \log n)$	$O(\log n)$	[38]
		d		$O(n^d \log n)$	$O(\log^{d-1} n)$	[22]
Link Center/ Link Diameter	No	2	simple polygon	$O(1)$	$O(n \log n)$	[152, 82, 40]
	Yes		simple rectilinear		$O(n)$	[127, 126]
k-Reachable Region	No	2	simple polygon	$O(1)$	$O(n \log n)$	[50]
	Yes		simple rectilinear		$O(kn)$	[14]

[a] The value $\alpha(n)$ appearing in the running time is the inverse of the Ackermann function.
[b] A '*' indicates that this dissertation is the source of the algorithm.

Table 1.2: A list of results in the area of link-distance path problems including the running times, approximation factors, and the source of the best known algorithm solving the problems.

Bicriteria Path Problems

Problem	Rect. Path?	Dim.	Restrictions on Obstacles	Link Approx.	Geodesic Approx.	Running Time	Source
Smallest Path	Yes	2	simple rectilinear	1	1	$O(n)$	[115]
						$O(n \log n)$; $O(\log n)$	[21] [a]
	No		simple polygon	2	$\sqrt{2}$	$O(n)$	[12]
Minimum Cost Path	Yes	2	rectilinear	1	1	$O(nt + n \log n)$	[160] [b]
Combined Metric Path		d				$O(n^d \log n)$	[22]
Minimum Total Turn k-link Path	No	2	simple polygon	1	–	$O(n)$	[139] [c]
			polygon with holes			$O(E^3 n\alpha(n) \log^2 n)$	
Min. Total Turn Short. Path				–	1	$O(En^2N^2)$	
Shortest k-Link Path	No	2	simple polygon	1	$1+\epsilon$	$O(n^3k^3 \log (Nk/\epsilon^{1/k}))$	[123] [c]
			polygon with holes	2	1	$O(kE^2)$	
	Yes		rectilinear	1	1	$O(k(nt + n \log n))$	[160] [b]
						$O(kn \log^2 n)$	[140]
Shortest Min. Link Path						$O(n \log n)$	[94]
Min. Link Shortest Path						$O(n \log^{3/2} n)$	[161]

[a] This result has $O(n \log n)$ preprocessing and $O(\log n)$ query time.

[b] Here t is the number of extreme edges among all obstacles, where an extreme edge is defined to be one with two convex corners.

[c] E is the number of edges in the visibility graph of the underlying polygon and N is the largest integer coordinate among the vertices.

Table 1.3: A list of results in the area of bicriteria link-distance path problems including the running times, approximation factors, and the source of the best known algorithm solving the problems.

Chapter 2

Applications

In this chapter a number of applications for minimum link-distance problems are listed. Some of these applications appear futuristic, but others have been applied to real life situations, and most have been researched in other contexts. Parts of this list were derived from Maheshwari, Sack, and Djidjev [112].

- **[VLSI]** A popular type of *very large scale integrated circuit (VLSI)* involves two layers of wires, with one set running horizontally and the other vertically. Signals on the circuitboard can run along either layer, and switching between the two directions is done through a point of where the two layers are connected, known as a *via*. A path is constructed as a series of vertical and horizontal lines connected with vias. Each via introduces some resistance along the path, so it becomes desirable to limit the number of vias. This then means finding a rectilinear path with a small link-distance. If there are several points which need to be connected, then the problem becomes one of minimizing the maximum number of vias between any two points within a tree connecting all points. This implies finding a minimum link-diameter spanning tree, where the link-diameter of a tree is defined to be the maximum among link-distances between any two points in the tree [117, 129, 146, 147, 160, 94, 161, 95, 96, 162].

- **[Graph Visualization]** The field of Graph Drawing is concerned with laying out a visual representation of a graph in the plane, such that the graph is represented clearly. One of the metrics that has been used to measure this objective is the number of bends in the entire layout [155].

- **[Homogeneous Modular Robots]** Homogeneous Modular Robots are comprised of many similar or identical, independently functioning entities. Alone, each unit has very limited capabilities. In one example these robots are cubical, and alone can only expand and contract in a dimension and connect to a neighbor. A large number of these working together, however, could be built into a complex entity which can reshape as is appropriate for a given situation. A fundamental operation within such an apparatus would be to fill an empty space at one point with a single cubical robot, while emptying the space occupied by a robot in another point. The speed with which such an operation can take place is limited by the minimum link-distance between the two points, through the space occupied by the cubes [54].

- **[Space Travel]** A messenger traveling through space may wish to deliver a message via radio to a number of recipients . If the messenger has only a weak radio signal, then he must travel close enough to each of these points in turn. Each turn he makes along this trip means extra fuel used, and thus it it desirable to minimize the number of turns. This then becomes the Minimum Link Traveling Salesman Problem with Neighborhoods [80].

- **[Milling out an Area / Mowing a Lawn]** A board which needs to be partially hollowed out can be carved with a milling machine whose bit carves a circular or square hole. Certain kinds of mills can quickly carve straight lines, so a series of straight lines is used to fill out the area. Minimizing the number of lines naturally becomes a desirable objective. This is known as the Milling Problem. The Lawn Mowing

Problem is similar to the Milling Problem, the difference being that lawn mower is not confined to stay within the bounds of the target area (It can mow parts of the neighbor's yard without penalty) [9, 6].

- **[Surveying an Area by Aircraft]** Consider a skier who is trapped under the snow by an avalanche, but who is wearing an emergency transmitter used for location. An aircraft pilot wants to run a series of flyovers through a large area where the skier is believed to be trapped, but the transmitter only broadcasts for a certain distance [149]. The pilot would like to make the fewest number of straight flyovers, while ensuring that the aircraft has traveled close enough to all points in the designated region to pick up the signal. This is similar to the Lawn Mowing Problem, although consecutive pairs of lines through the target area may not have the same connectivity requirements.

- **[Positioning Mirrors]** A laser beam which must be sent through a crowded apparatus can be reflected off a series of mirrors in order to reach its destination. Minimizing the number of mirrors is a natural objective and is equivalent to the Minimum Link-Distance Problem.

- **[Communication Systems]** A series of cell phone towers must be located in such a way so as to provide cellular service to certain desirable locations. Two towers can only communicate if they have a direct line of sight. Mountains and buildings may stand in the way, so the towers must be strategically positioned, without incurring too much total cost to build the towers. This problem can be modeled as a variety of link-distance problems. If only two locations need to be connected, then this is the Minimum Link Path Problem. If there is an already established central station, then it can be modeled as the Single-Source Minimum Link Path Problem. If a site for a central processing station needs to be chosen, then this is the Link Center problem.

If a large number of locations need to be connected efficiently, then this becomes a Minimum Link Spanning Tree Problem [141].

- **[Telescoping Manipulators]** A robotic arm needs to reach inside a polygon while respecting the boundaries of that polygon. A well chosen point of entry can reduce the number of "elbows" which need to be built into the arm, or equivalently, the number or straight line segments needed to reach the points in the interior or the polygon. This can be modeled as the Link Center Problem when choosing a fixed entry point, or as the Single-Source Minimum Link Path Problem when routing the arm through the polygon[87].

- **[Pin Gate Positioning]** The *pin gate* is the location where a liquid is poured or injected into a mold. It is desirable to both minimize the maximum Euclidean distance and the maximum link-distance to all points within the mold from the pin gate, hence a bicriteria center could be useful here [25].

- **[Traffic Routing]** In choosing a route for an automobile, it may be more costly to turn at intersections than to continue straight, and perhaps one direction of turn (left or right) is more costly than the other. This might be modeled as a link-distance problem, perhaps assigning different weights to different directions [19].

In many applications, both distance and number of turns can contribute to the cost function. Problems such as the Combined Metric [22], Minimum Cost [160], Shortest k-Link Path [123, 140], Shortest Minimum Link Path [94], Minimum Link Shortest Path [161], and other Bicriteria Problems [139] address this objective. The work in this dissertation is hopefully a step towards understanding this more general problem.

Chapter 3

The Minimum Link Traveling Salesman Problem

The problem of traversing a set of points in the order that minimizes the total distance traveled, famously knows as the *Traveling Salesman Problem*, is one of the most well-studied problems in combinatorial optimization, inspiring large textbooks devoted entirely to the problem [92, 65]. It has many applications [100, 57, 59, 113, 74], and has been a testbed for many of the most useful ideas in algorithm design and analysis. The usual metric, minimizing the total distance traveled, is an important one, but many other metrics including maximum total distance (known as the *maximum TSP (MAX-TSP)*) [53, 90, 89, 66, 67], minimum latency (sometimes called the *traveling repairman problem* or the *deliveryman problem*) [1, 24], minimum longest edge length (also known as the *bottleneck TSP (BTSP)*) [58, 28], maximum shortest edge length (known as the *maximum scatter TSP*) [8], minimum total turn (the *angular-metric TSP*) [2], and minimum total distance to a point within each of a set (*TSP with neighborhoods (TSPN)*) [10, 62, 63, 43, 143] are also of interest.

In this chapter, we apply the metric of minimizing the link-distance of the tour. This problem is known as the *Minimum Link Traveling Salesman Problem*, or simply as the

Minimum Link Tour Problem. The objective is to find a tour, π, covering a given set of input points, P, in the Euclidean plane consisting of a series of straight lines, $\pi = \{l_1, l_2, \ldots, l_k\}$, such that each point $p_i \in P$ lies along one of the lines, $l_j \in \pi$, and the number of lines, k, is minimized. We give some approximation algorithms for rectilinear and non-rectilinear variants of this problem, and also prove the non-rectilinear problem to be NP-Complete. These results are published separately [150], except for the NP-Completeness result which has been independently proven by a different method. This was published in 2003 [11].

In the case of an arbitrary set of n points in the Euclidean plane, an $O(\log(\min(z, c)))$-approximation algorithm, where z is the maximum number of collinear points and c is the minimum number of lines that can be used to cover all n points, is given. In the worst case $\min(z, c)$ can be as big as $n/3$, but it will often be smaller.

We also study some restricted cases and find better approximation ratios. We introduce the *Rectilinear Minimum Link Traveling Salesman Problem*, in which the lines of the tour are restricted to being either horizontal or vertical, and we give a 2-approximation algorithm for this problem. The algorithms for both cases involve forming a relaxation which we call the *line cover* problem, where the line cover is the minimum-sized set of lines covering all the input points. The differences in the two algorithms arise in the choice of potential lines to include in the cover and the ability to approximate the resulting line cover instance. For the general case, we obtain a set-cover problem. Our logarithmic approximation ratio results from the logarithmic approximation ratio of set-cover.[26, 35, 42]. For the rectilinear case, we obtain a bipartite vertex cover problem, which can be solved in polynomial time [78, 73].

Finally, we consider a special case of the rectilinear minimum link traveling salesman problem in which we have the restriction that no two points in the input are allowed to have the same x- or y-coordinate. This allows us to study carefully an aspect of the problem which the previous approximation algorithms ignore. Once they find a line cover, the

other algorithms link the lines together in a fairly straightforward way. For this problem, however, the minimum sized line cover is equal to the number of input points, so the line cover gives us essentially no helpful information. Instead we focus on how to carefully link together non-collinear points into a tour. For this case, we give an algorithm that finds a tour which makes at most two turns more than the optimal tour. Thus we have an approximation algorithm with an additive, rather than a multiplicative error bound.

Beyond the additive error bound, our algorithm for this problem introduces several interesting algorithmic techniques. We introduce two different ways to decompose a set of points in the Euclidean plane. We call these decompositions a *9-division* and a *4-division*. We then show that any set of points can either be decomposed into a 9-division or a 4-division. Guided by these decompositions, we repartition the points into a set of points that are monotonically increasing, a set of points that are monotonically decreasing, and a set of points that fall on the perimeters of a set of nested boxes. Using this second decomposition, we are able to find a tour that uses at most two turns more than the optimal tour. We believe that these decompositions may be of independent interest.

We show that the general minimum link traveling salesman problem is NP-hard in Chapter 3.6, by reducing the Line Cover problem to it [116]. We do not know whether the remaining problems considered in this chapter are NP-hard, but we would not be surprised if the non-collinear rectilinear variant (c.f. Section 3.2) is actually solvable in polynomial time.

3.1 Background and Related Problems

Before discussing the problems under consideration, it is worthwhile to mention some previous results on the Traveling Salesman Problem itself. It is also useful to review some of the problems which will be used to help solve the minimum link traveling salesman

problem.

In this section the most important of those results are revisited. First both non-geometric and the geometric variants of the Traveling Salesman Problems are revisited. Second, results on the *Set Cover (SC)* problem are discussed. This problem figures prominently in both the algorithm and and NP-completeness proof for the Non-Rectilinear Minimum Link Traveling Salesman Problem (see Sections 3.4 and 3.6). Third, the *Bipartite Vertex Cover Problem (BVC)*, which is used in the solution to the Rectilinear Minimum Link Traveling Salesman Problem (see Section 3.3), is discussed. Finally, the *Line Cover Problem (LC)* is described, and relevant results are mentioned. This problem is a useful intermediate problem in analyzing both rectilinear and non-rectilinear versions of the Minimum Link Traveling Salesman Problem.

3.1.1 Traveling Salesman Problem

The Traveling Salesman Problem has a rich history, and a complete discussion of it would require many pages. Here, some of the most prominent results on that problem are reviewed.

First, consider the regular non-geometric traveling salesman problem in a graph with edge weights. This problem is well known to be NP-Complete, and without the triangle inequality, it is even hard to approximate within a constant factor, as shown by Sahni and Gonzalez in 1976 [144].

Under the triangle inequality, it remains NP-Complete, but Christofides' algorithm from 1976 gives a $\frac{3}{2}$-approximation [34]. The problem is also known to be Max SNP-Hard, meaning that no PTAS is likely to exist. However, in a 1993 result of Papadimitriou and Yannakakis, it has been shown that if all edge weights are restricted to the set $\{1, 2\}$ then the traveling salesman problem can be approximated to within an factor of $\frac{7}{6}$, although this

problem remains Max SNP-Hard [137].

In the geometric traveling salesman problem, the nodes correspond to points in the Euclidean plane, and edge lengths between nodes are the corresponding Euclidean distances. This problem is still NP-Complete [136, 56], however the algorithms of Arora [13] and of Mitchell [119] each give a PTAS for the problem.

For the *longest tour (MAX-TSP)*, where the objective is to find a tour having the *maximum* length, the non-geometric version is also Max SNP-Hard [89, 137], although a close $\frac{5}{7}$-approximation algorithm [66], and a closer randomized $\frac{25}{33}$-approximation algorithm [67] have each been achieved by Hassin and Rubinstein.

In the case of the geometric MAX-TSP, when the points are in R^d for some fixed d and distances are computed according to some polyhedral norm, the problem is solvable in polynomial time [16]. This includes the rectilinear, or L_1, norm but not the non-polyhedral Euclidean, or L_2, norm. The problem becomes NP-Hard under the Euclidean norm in R^d for $d \geq 3$, although interestingly enough its hardness remains an open problem when $d = 2$ [52]. A more complete overview of the geometric MAX-TSP was published by Barvinok, Fekete, Johnson, Tamir, Woeginger, and Woodroofe in 2003 [15].

Some other geometric objectives such as the traveling repairman problem, where the objective is to minimize the sum of the delay times to reach each of the destinations[24], and the maximum scatter TSP, where the objective is to find a tour having the *longest* minimum edge length [8], have constant-factor approximation schemes. A survey of polynomial time solvable geometric and non-geometric cases of the TSP was published by Burkard, Deĭneko, van Dal, van der Veen, and Woeginger in 1998 [27].

In contrast, the best known approximation algorithm for the Angular-Metric TSP, where the objective is to minimize the sum of the angles of all turns along the tour, is $O(\log n)$ [2], and it is known that TSP with Neighborhoods, where the objective is to visit any point within each of a set of regions, cannot be approximated to within any constant factor [143].

3.1.2 Set Cover

A *set system* $(S, \mathcal{F})$ is defined to be a set $S = \{s_1, s_2, \ldots, s_n\}$, together with a family of sets $\mathcal{F} = \{f_1, f_2, \ldots, f_m\}$, each of whose elements are drawn from S. That is, for each set $f_i \in \mathcal{F}$ we have $f_i \subseteq S$.

Given a set system $(S, \mathcal{F})$, the problem of choosing a smallest subset $F \subseteq \mathcal{F}$ such that for each element $s_i \in S$ there exists some $f_j \in F$ such that $s_i \in f_j$, is well known as the Set Cover Problem. It has been studied extensively since the problem was shown to be NP-Complete by Karp in 1972 [81].

In 1974, Johnson introduced the greedy heuristic for the Set Cover problem as part of a collection of approximation algorithms [76]. Independently, Lovász described a greedy algorithm for finding a vertex cover of a hypergraph in 1975 [110].

The greedy method involves repeatedly choosing the set $f \in \mathcal{F}$ or vertex which makes the most progress toward covering all elements. When the first choice is made this is simply the largest set, or the vertex with the highest degree. Subsequent choices are the set or vertex which includes the largest number of elements not included in a previous choice.

In each case Johnson and Lovász show that their greedy method results in a solution which is within a factor of $\mathcal{H}(D)$ of optimal, where $\mathcal{H}(D)$ is the Harmonic Series $\sum_{i=1}^{D} \frac{1}{i}$, and D is the size of the largest set or the largest vertex degree, thereby yielding an approximation ratio of $O(\log D)$. In fact, it is not difficult to see that the two problems are equivalent.

In 1979 Chvátal extended the use of the greedy heuristic to the weighted set cover problem [35]. In this problem, each set $f_j \in \mathcal{F}$ has a weight w_j, and the objective is to minimize $\sum_{j|f_j \in F} w_j$, the total weight of the sets chosen. Chvátal showed that the $\mathcal{H}(D)$ approximation factor continues to hold when the greedy method is used with the weighted problem.

In 1982, Hochbaum used a different method to find an f-approximation for set cover, where f is the maximum number of sets covering a single element [71].

In 1993, Goldschmidt, Hochbaum, and Yu improved the approximation bound on the greedy algorithm to $\mathcal{H}(D) - 1/6$ [60], and in 1997 Duh and Fürer improved this to $\mathcal{H}(D) - 1/2$ [42].

In 1994, Lund and Yannakakis showed that Set Cover is hard to approximate to within a ratio of $\frac{1}{4}\log n$, as part of a collection of hardness of approximation results [111]. In 1998 Feige showed that set cover cannot be approximated to within a factor of $(1 - o(1)) \ln n$ [51]. These results come under reasonable assumptions about the existence of certain kinds of problems in NP.

In 1996 Slavík gave a tight analysis of the greedy method, showing that its approximation ratio is exactly $\ln D - \ln \ln D + \Theta(1)$ [148]. These analyses make the greedy heuristic appear quite close to optimal, although small constant factor improvements might still be made.

The logarithmic approximation of set cover will be used to find a logarithmic approximation of the minimum link tour in Section 3.4. Although the minimum link tour is also shown to be NP-Complete in Section 3.6, it remains open whether it is hard to approximate within a constant factor.

The VC-Dimension of Set Systems

An attribute of Set Cover instances known as the *VC-Dimension* has been used to improve on the approximation bound for certain classes of set systems. The VC-Dimension gets its name from a 1971 paper of Vapnik and Červonenkis [159]. It is defined as follows.

Given a set system $(S, \mathcal{F})$, consider a subset $Y \subseteq S$. Let $\mathcal{G}$ be the family of sets consisting of all $f \cap Y$ such that $f \subseteq \mathcal{F}$. It is said that Y is *shattered* by $\mathcal{F}$ if $\mathcal{G} = 2^Y$. The VC-Dimension of $(S, \mathcal{F})$ is the size of the largest subset $Y \subseteq S$ which is shattered by $\mathcal{F}$,

or zero if no such subset exists [26, 159].

In 1995 Brönnimann and Goodrich showed that if a set system has VC-Dimension v, and if its optimal set cover has size z, then the set cover can be approximated to within a factor of $O(v \log(vz))$ of optimal in polynomial time. Thus, a set system with a constant VC-Dimension can be approximated to within $O(\log z)$ of optimal, which is often an improvement on the above mentioned results [26].

This result will be used to improve the approximation ratio of an algorithm finding a minimum link tour in Section 3.4.

3.1.3 Bipartite Vertex Cover

Given an graph $G = (V, E)$, an edge $e \in E$ is said to be covered by vertex $v \in V$ if v is one of the endpoints of e. The *vertex cover* of G is the smallest set of vertices $W \subseteq V$ such that every edge in E is covered by some vertex in W.

Here we will be concerned with the special case when G is a *bipartite* graph. A graph is bipartite if its vertices can be partitioned into two distinct sets L and R such that every edge in E has one endpoint in L and one endpoint in R.

In a classic result from 1931 attributed to König [88], and independently to Egerváry [49], it is known that the size of a minimum vertex cover of a bipartite graph is equivalent to the size of a maximum matching on that same graph.

Further progress was made in 1962, when Ford and Fulkerson demonstrated in Chapter 2, Section 5 of [78] that a maximum flow through a bipartite graph can be used to find a maximum matching for that graph, and further that its dual, the minimum cut, in turn yields a minimum vertex cover of the graph.

In 1973, it was shown by Hopcroft and Karp that a maximum matching can be found in a bipartite graph in $O(|E|\sqrt{|V|})$ time [73]. This can easily be translated into a minimum

vertex cover using, for example, Algorithm 1.

Algorithm 1 translates such a matching into a vertex cover of the same size. This size taken together with the König-Egerváry theorem, makes the resulting vertex cover optimal.

Algorithm 1 FIND-BIPARTITE-VC(G, M)

Given: A bipartite graph $G = (V, E)$, and a maximum matching M on G. The vertices of G are divided into two sets $V = L \cup R$, and the edges of G are exclusively between L and R.

Return: A minimum vertex cover of G.

```
1: for all ( vertices v ∈ L ∪ R ) do
2:    if ( v ∈ L ∩ M ) then
3:       v.color ← RED
4:    else if ( v ∈ R ∩ M ) then
5:       v.color ← BLUE
6:    else if ( v ∉ M ) then
7:       v.color ← BLACK // This vertex is not in the cover
8: while ( ∃(u, v) ∈ E with u.color ∈ {BLACK, WHITE} and v.color ∈ {RED, BLUE}
   ) do
9:    if ( u.color = BLACK ) then
10:      v.color ← WHITE // This vertex is in the cover
11:   else if ( u.color = WHITE ) then
12:      v.color ← BLACK // This vertex is not in the cover
13: for all ( vertices v ) do
14:   if ( v.color = RED ) then
15:      v.color ← WHITE
16:   else if ( v.color = BLUE ) then
17:      v.color ← BLACK
18: return all v ∈ L ∪ R with v.color = WHITE
```

3.1.4 Line Cover

A point p in the Euclidean plane is said to be covered by a line l if p lies somewhere along l. Given a set of points $P = \{p_1, p_2, \ldots, p_n\}$ in the plane, the *Line Cover Problem* asks what is the smallest number, k of lines $L = \{l_1, l_2, \ldots, l_k\}$ such that each point $p \in P$ is covered by some line $l \in L$.

In 1982, this problem was shown to be NP-complete by Megiddo and Tamir. In their

paper they refer to it as the *Point Cover Problem*, but here it is called the Line Cover Problem, since this name is analogous to Vertex Cover. (In Vertex Cover it is the vertices doing the covering. Here it is the lines doing the covering.) [116].

In 1991, Hassin and Megiddo showed that the Rectilinear Line Cover Problem can be solved in $O(n^{1.5})$ time by reducing it to the Bipartite Vertex Cover problem, as proposition 2.2 of [68]. They use earlier methods shown in 1962 by Ford and Fulkerson [78], but in the context of the Line Cover Problem and using the matching algorithm of Hopcroft and Karp from 1973 [73].

The Line Cover Problem is closely related to the Minimum Link Traveling Salesman Problem, since the line segments of any tour form a line cover. This similarity is used in Section 3.6 to prove the NP-Completeness of the Minimum Link Tour Problem. The similarity was also used by Arkin, Mitchell, and Piatko in 2003 to prove the same result [11].

We give formal definitions of the Line Cover problem here:

Definition 3.1.1 (Line Cover Problem (LC))

Given: *A set of points $P = \{p_1, p_2, \ldots, p_n\}$ in the Euclidean Plane.*

Find: *A set of lines $L = \{l_1, l_2, \ldots, l_k\}$ such that each $p_i \in P$ falls along some $l_j \in L$ and such that k is minimized. Let $LC(L)$ be the optimal value for k.*

Definition 3.1.2 (Rectilinear Line Cover Problem ($\square LC$))

Given: *A set of points $P = \{p_1, p_2, \ldots, p_n\}$ in the Euclidean Plane.*

Find: *A set of horizontal and vertical lines $L = \{l_1, l_2, \ldots, l_k\}$ such that each $p_i \in P$ falls along some $l_j \in L$ and such that k is minimized. Let $\square LC(L)$ be the optimal value for k.*

3.2 Preliminaries

In this section we introduce three versions of the Minimum Bends TSP. We also define the Line Cover problem and its rectilinear variant, both of which will be useful subroutines in our algorithms.

Throughout this chapter, we use the convention that a point p_i has x and y coordinates x_i and y_i respectively. When a point p_i falls on line l_j, we will say that line l_j *covers* point p_i.

We will be concerned with approximation algorithms, and will define a ρ-approximation algorithm for a minimization problem to be one which, in polynomial time, finds a solution of value $\rho OPT + O(1)$, where OPT is the value of the optimal solution to the problem.

3.2.1 Segmented Tours and Cycle Covers

In this chapter, we will consider traveling salesman tours in the plane. Our tours will differ from conventional tours in that the endpoints of their segments need not be at input points.

Definition 3.2.1 (Segmented Tour (S-Tour)) *Given a set of points $P = \{p_1, p_2, \ldots, p_n\}$ in the Euclidean plane, define an S-Tour over P to be a sequence of line segments $\pi = l_0, l_1, \ldots, l_{m-1}$ such that:*

1. *There exists a set of points $Q = \{q_0, q_1, \ldots, q_{m-1}\}$ such that the endpoints of l_i are q_i and $q_{i+1 \bmod m}$*
2. *Each point $p_i \in P$ falls along some line $l_j \in \pi$.*

Definition 3.2.2 (Segmented Cycle Cover (S-Cycle Cover)) *Given a set of points $P = \{p_1, p_2, \ldots, p_n\}$ in the Euclidean plane, define an S-Cycle Cover over P to be a sequence of line segments $\pi = l_0, l_1, \ldots, l_{m-1}$ such that:*

1. *There exists a set of points $Q = \{q_0, q_1, \ldots, q_{m-1}\}$ such that the endpoints of l_i are q_i and q'_i, where the sequence $\{q'_0, q'_1, \ldots, q'_{m-1}\}$ is a permutation of Q.*

2. *Each point $p_i \in P$ falls along some line $l_j \in \pi$.*

Definition 3.2.3 (Rectilinear Segmented Tour (□S-Tour)) *Given a set of points $P = \{p_1, p_2, \ldots, p_n\}$ in the Euclidean plane, define a □S-Tour over P to be an S-Tour $\pi = l_0, l_1, \ldots, l_{m-1}$ over P with the following additional property:*

3. *Each $l_j \in \pi$ is a horizontal or vertical line segment.*

Definition 3.2.4 (Rectilinear Segmented Cycle Cover (□S-Cycle Cover)) *Given a set of points $P = \{p_1, p_2, \ldots, p_n\}$ in the Euclidean plane, define a □S-Cycle Cover over P to be an S-Cycle Cover $\pi = l_0, l_1, \ldots, l_{m-1}$ over P with the following additional property:*

3. *$l_j \in \pi$ is a horizontal line segment if j is even and a vertical line segment if j is odd (or vise-versa).*

3.2.2 Translating a Point Sequence into a Tour

Solutions to TSP problems are traditionally given as an ordering on the set of input points. Although a rectilinear segmented tour is defined as a sequence of line segments, it can also be uniquely represented as a sequence of points and a starting direction, making its representation closer to that of traditional solutions to the TSP. Given a set of points $Q = \{q_0, q_1, \ldots, q_{m-1}\}$, and a starting direction d, a natural rectilinear tour associated with Q and d traverses the points in order. To go from point q_i to point q_{i+1}, greedily choose the path that minimizes the number of line segments needed to travel to q_{i+1}, having approached q_i in a particular direction. This could involve zero, one, or two bends, depending on the locations of the points and the direction of approach.

The formulation is not as obvious when translating an ordering of points into a non-rectilinear tour. In this case, dynamic programming can be used to perform the translation in linear time. Consider that, given an ordering on the points, each point may lie along the same line segment as its predecessor, its successor, neither, or both (if possible). A dynamic programming algorithm can then compute the optimal link-distances to q_{i+1} in each of these situations, given the corresponding link-distances to q_i.

3.2.3 Problem Definition

We now define the main problem of this chapter along with two restricted versions. In the *Minimum Link TSP ($MLTSP$)*, we are given a set of points $P = \{p_1, p_2, \ldots, p_n\}$ in the Euclidean Plane, and we wish to find an S-Tour $\pi = l_0, l_1, \ldots, l_{k-1}$ of link-distance k over P such that k is minimized. We will let $MLTSP(P)$ denote the optimal value for k. Similarly, in the *Rectilinear Minimum Link TSP ($\square MLTSP$)*, we wish to find a $\square$S-Tour $\pi = l_0, l_1, \ldots, l_{k-1}$ over P such that k is minimized. We will let $\square MLTSP(P)$ denote the optimal value for k. In the *Non-Collinear Rectilinear Minimum Link TSP (NC-$\square MLTSP$)*, our points have the further restriction that for any $p_i, p_j \in P$ with $i \neq j$ we have $x_i \neq x_j$ and $y_i \neq y_j$ (in the future we will call this the *non-collinearity property*). We wish to find a $\square$S-Tour $\pi = l_0, l_1, \ldots, l_{k-1}$ over P such that k is minimized. We will let NC-$\square MLTSP(P)$ denote the optimal value for k. For all these problems, the number of bends in a tour is equivalent to the number of line segments. Thus our objective, minimizing bends, is characterized by minimizing k, the number of line segments in the tour.

Definition 3.2.5 (Minimum Link TSP ($MLTSP$))

Given: *A set of points $P = \{p_1, p_2, \ldots, p_n\}$ in the Euclidean Plane.*

Find: *An S-Tour $\pi = l_0, l_1, \ldots, l_{k-1}$ over P such that k is minimized. Let $MLTSP(P)$ be the optimal value for k.*

We will be especially concerned with the case when the tour is restricted to using horizontal and vertical lines, and with a restricted case of this problem, in which no two points lie on the same horizontal or vertical line.

Definition 3.2.6 (Rectilinear Minimum Link TSP ($\square MLTSP$))

Given: *A set of points $P = \{p_1, p_2, \ldots, p_n\}$ in the Euclidean Plane.*

Find: *A $\square$S-Tour $\pi = l_0, l_1, \ldots, l_{k-1}$ over P such that k is minimized. Let $\square MLTSP(P)$ be the optimal value for k.*

Definition 3.2.7 (Non-Collinear Rectilinear Minimum Link TSP (NC-$\square MLTSP$))

Given: *A set of points $P = \{p_1, p_2, \ldots, p_n\}$ in the Euclidean Plane such that for any $p_i, p_j \in P$ with $i \neq j$ we have $x_i \neq x_j$ and $y_i \neq y_j$ (in the future we will call this the non-collinearity property).*

Find: *A $\square$S-Tour $\pi = l_0, l_1, \ldots, l_{k-1}$ over P such that k is minimized. Let NC-$\square MLTSP(P)$ be the optimal value for k.*

3.3 A 2-approximation algorithm for $\square MLTSP$

In this section we give a 2-approximation algorithm for the Rectilinear version of the Minimum Link Traveling Salesman Problem ($MLTSP$) As described in Section 3.2, we are given an arbitrary set of points in the Euclidean plane, and we are looking for a rectilinear tour which minimizes the number of turns in the tour. The Rectilinear Line Cover problem will be used in our approximation algorithm for $\square MLTSP$. Hassin and Megiddo showed that this problem can be solved in $O(n^{1.5})$ time by reducing it to the Bipartite Vertex Cover problem[68].

Our algorithm first computes an optimal rectilinear line cover on the set of input points. We then convert this to a rectilinear tour having no more than twice as many turns as there

are lines in the optimal rectilinear line cover. The resulting tour is a 2-approximation since the number of lines in the rectilinear line cover instance defines an obvious lower bound on the $\square MLTSP$ instance.

Algorithm 2 SMALLEST-ENCLOSING-RECTANGLE(P)

Given: A set of points $P = \{p_1, p_2, \ldots, p_n\}$ in the plane, where $p_i = (x_i, y_i)$.

Return: $maxx$, $maxy$, $minx$, and $miny$, the points with the largest and smallest x and y-coordinates. Note, some of these may be the same point.

1: $maxx \leftarrow \max_{x_i}(p_i)$
2: $maxy \leftarrow \max_{y_i}(p_i)$
3: $minx \leftarrow \min_{x_i}(p_i)$
4: $miny \leftarrow \min_{y_i}(p_i)$
5: **return** $\{maxx, maxy, minx, miny\}$

Algorithm 3 FIND-RECTILINEAR-MINLINKTOUR(P, L)

Given: A set of points P, and an optimal Rectilinear Line Cover $L = H \cup V$ over P, consisting of horizontal lines $H = h_1, h_2, \ldots, h_x$ and vertical lines $V = v_1, v_2, \ldots, v_y$.

Return: A 2-approximation of the Rectilinear Minimum Link Tour of P.

1: $\{maxx, maxy, minx, miny\} \leftarrow$ SMALLEST-ENCLOSING-RECTANGLE(P)
2: $v_L \leftarrow v_0 \leftarrow$ the vertical line through $minx$
3: $v_R \leftarrow v_{y+1} \leftarrow$ the vertical line through $maxx$
4: $h_B \leftarrow h_0 \leftarrow$ the horizontal line through $miny$
5: $h_T \leftarrow h_{x+1} \leftarrow$ the horizontal line through $maxy$
6: **for all** ($i \leq x$) **do**
7: $\quad l_i \leftarrow$ the line segment between h_i and h_{i+1} along v_L.
8: $\quad r_i \leftarrow$ the line segment between h_i and h_{i+1} along v_R.
9: **for all** ($i \leq y$) **do**
10: $\quad b_i \leftarrow$ the line segment between v_i and v_{i+1} along h_B.
11: $\quad t_i \leftarrow$ the line segment between v_i and v_{i+1} along h_T.
12: $\pi_H \leftarrow r_0, h_1, l_1, h_2, r_2, h_3, ..., h_x$
13: $\pi_V \leftarrow t_0, v_1, b_1, v_2, t_2, v_3, ..., v_y$
14: $\pi \leftarrow \pi_H, l_x, \pi_V, b_y$ // Substitute for l_x and/or b_y as necessary
15: **return** π

Lemma 3.3.1 *Given a set of points $P = \{p_1, p_2, \ldots, p_n\}$ in the Euclidean plane, if $\square LC(P) = k$, then $\square MLTSP(P) \leq 2k + O(1)$.*

Proof. Let $L = H \cup V$ be an optimal rectilinear line cover over the points in P, consisting of x horizontal lines $H = h_1, h_2, \ldots, h_x$ and y vertical lines $V = v_1, v_2, \ldots, v_y$. Define a bounding box B such that all points in P are contained within the boundaries of B. Construct a path through H as follows.

Beginning at h_1 connect h_1 to h_2 along the maximum boundary of B. Next connect h_2 to h_3 along the minimum boundary of B. Continue in this way, connecting h_i to h_{i+1} along either the maximum or minimum boundary, depending on the parity of i, for all $i \in [1, x-1]$. The total number of lines used to cover H is then $2x - 1$. Construct an analogous path covering V and containing $2y - 1$ lines. Joining the horizontal and vertical sections at both ends adds 4 more lines for a total of $2x-1+2y-1+4 = 2k+O(1)$ lines. Figure 3.1 shows a sample tour, and Algorithm 3 gives pseudocode for this construction. □

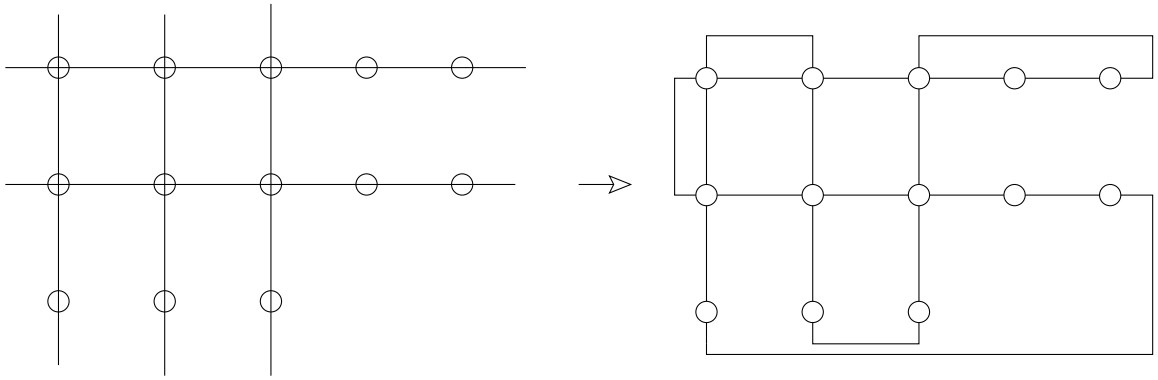


Figure 3.1: A sample tour formed by $\square MLTSP$

The lemma is tight in the sense that there exist sets of points P for which $\square MLTSP(P) \geq 2\square LC(P)$, as shown in the following Lemma.

Lemma 3.3.2 *For any n_0 there exists a set of $n \geq n_0$ points $P = \{p_1, p_2, \ldots, p_n\}$ such that $\square MLTSP(P) \geq 2\square LC(P)$.*

Proof. Given n_0, consider G_{n_0} the complete $n_0 \times n_0$ grid of points. There exists a line cover over G_{n_0} containing n_0 lines. Now, it remains to show that any rectilinear tour of G_{n_0}

must contain at least $2n_0$ links. First note that any rectilinear tour must alternate between horizontal and vertical links. Thus it will be sufficient to show that any tour of G_{n_0}, must contain at least n_0 horizontal, or at least n_0 vertical links. We prove that next.

Assume, by way of contradiction, that there exists a tour of G_{n_0} with at most $n_0 - 1$ horizontal links, and at most $n_0 - 1$ vertical links. The horizontal links can cover at most $n_0 - 1$ rows of points, leaving at least one uncovered row of n_0 points, each of which has a unique x-coordinate. The remainder of the tour consists of at most $n_0 - 1$ vertical links. Since each of the remaining n_0 points has a unique x-coordinate, they cannot all be covered by these remaining lines. Thus, there must either be at least n_0 vertical lines, or n_0 horizontal lines in any tour of G_{n_0}. □

Combining this lemma with Hassin and Megiddo's algorithm we obtain:

Theorem 3.3.3 *A 2-approximation for the Rectilinear Minimum Bends Traveling Salesman Problem can found in $O(n^{1.5})$ time.*

Proof. The approximation bound follows from the trivial lower bound of $\square LC(P) \leq \square MLTSP(P)$ and the upper bound of $\square MLTSP(P) \leq 2\square LC(P) + O(1)$. shown in Lemma 3.3.1. The running time follows from the $O(n^{1.5})$ running time of $\square LC$ shown by Hassin and Megiddo, and the algorithm described above which, given a $\square LC$, constructs a tour in $O(n)$ time. □

3.3.1 Rectilinear Cycle Cover from Rectilinear Line Cover

For several TSP problems, a cycle cover relaxation is a useful algorithmic tool. A Rectilinear Cycle Cover Problem analogous to $\square MLTSP$ was defined in Section 3.3.1. Note that from a given rectilinear line cover, it is possible to optimize the rectilinear cycle cover using those lines by finding an appropriate matching between vertical and horizontal endpoints.

This, however, does not give us an algorithm for finding the optimal rectilinear cycle cover, as the optimal rectilinear cycle cover may not use the lines from the optimal rectilinear line cover.

On the other hand, it is possible to show that a given rectilinear cycle cover can be converted into a rectilinear path, using no more than $5/4$ times as many turns, plus one additional bend to make the path into a tour. Given the above mentioned difficulty with finding the optimal rectilinear cycle cover, we do not yet know how to make use of this fact.

Theorem 3.3.4 *Given a set of points P, and a rectilinear cycle cover over those points containing k links, we can find a rectilinear tour of P having at most $\frac{5}{4}k + 1$ links.*

Proof. Let C be a cycle cover over P containing j cycles $c_1, c_2, \ldots, c_j$ and k links. Note that $j \leq k/4$ since each cycle contains at least four links. Let B be a bounding box around the points of P. Construct a tour as follows.

Begin with any horizontal link in the cycle c_1. Follow the cycle c_1, and continue to follow the last vertical link of c_1 until it intersects a horizontal boundary of B. Next, extend any vertical link of c_2 until it intersects the same boundary of B. From the last link of c_1, follow the boundary of B to this, the first link of c_2. Follow the cycle c_2, and continue to follow the last horizontal link of c_2 until it intersects a vertical boundary of B. Analogously proceed along the boundary of B to the first link of c_3. Continue to alternate between following the remaining cycles and the boundary of B, until all cycles have been covered. Finish by connecting the final segment of c_j to the first segment of c_1, which may require one or two extra links, depending on the parity of j. The resulting tour uses only the links of C, plus at most $j + 1$ links along the boundary of B. Since $j \leq k/4$, this tour has at most $\frac{5}{4}k + 1$ links. □

3.4 An $O(\log n)$-approximation algorithm for $MLTSP$

We now turn to the general minimum link TSP problem. Consider what happens if we try to apply the algorithm of the previous section to this problem. One approach is to formulate a line cover problem as before, constraining the candidate lines to be those which cover maximal collinear subsets of the input points, together with the degenerate "lines" formed by single points. The resulting line cover problem can then be solved, and this can be used to obtain a tour, paying roughly a factor of 2 in the process. The only problem in this approach is that the resulting line cover problem is no longer equivalent to a bipartite vertex cover problem. Instead, it is now a set cover problem, and so our approximation bound will not be as close.

The details of the algorithm appear as Algorithm 4. In the first seven lines, the set T is computed, It contains all sets of points which might form lines in a line cover, including all lines going through two or more points, and the "line" going through only one input point whose direction is arbitrary. Hence singleton points are included as degenerate lines. The set T can be computed in $O(n^2)$ time, as shown in lines 1 through 7 of Algorithm 4.

A set-cover instance is now computed as follows. The elements of the set system (P, T) are the initial input points P, while the sets are T, the sets of points lying along lines which could be in a line cover. The optimal set cover of this set system is a lower bound on the optimal tour. Line 8 of Algorithm 4 finds a set cover over this set system.

Then if T' is any line cover over the points P, ordering the lines of T' arbitrarily, and connecting the two endpoints of each two consecutive segments with an additional line segment, forms a tour having twice as many line segments as the line cover. The code in lines 9 through 13 achieves this.

Theorem 3.4.1 *Algorithm 4, given a set of points P, computes a tour over P which is a 2ρ-approximation of the minimum link tour, where ρ is the approximation bound for Set*

Algorithm 4 FIND-MINLINKTOUR(P)

Given: A set of points P in the plane.
Return: An $O(\log n)$-approximation of the minimum link tour of P.

```
1: T ← ∅; k ← 0
2: for all ( p_i ∈ P ) do
3:    for all ( p_j ∈ P ) do
4:       S ← the set of all points in P along the line through (p_i, p_j)
5:       T ← T ∪ {S} // T is a set of sets of points
6: for all ( p_i ∈ P ) do
7:    T ← T ∪ {{p_i}} // T includes every set containing just a singleton point
8: T' ← SET-COVER(T, P) // This returns an O(log n)-approximation of set cover
9: for all ( S ∈ T' ) do
10:   if ( S contains only the singleton point p ) then
11:      q_2k ← q_2k+1 ← p; k ← k + 1
12:   else
13:      {q_2k, q_2k+1} ← The two extremum points in S; k ← k + 1
14: for all ( i < 2k ) do
15:   l_i ← the line segment (q_i, q_(i+1) mod 2k)
16: π ← l_0, l_1, ..., l_2k-1
17: return π
```

Cover.

Proof. Any tour over the points must use at least as many lines as there are in the optimal line cover. This algorithm uses 2ρ times as many lines. □

In general, the best set cover approximation is $\ln n$ [35, 72]. However, in the case when each set is of size no more than z, the approximation ratio is roughly $\ln z$, and tighter bounds are known for small values of z [42].

3.4.1 Improvements on the Approximation Ratio

Recall that in Section 3.1.2 it was mentioned that the approximation ratio of Set Cover can be improved if the VC-Dimension of the set system under consideration is constant. Lemma 3.4.2 shows that the VC-Dimension of the set system given to the SET-COVER function in Algorithm 4 is two, assuming there are at least three points. Therefore, the

improvement on the approximation bound can be applied.

Lemma 3.4.2 *The VC-Dimension of the set system (P, T), where P is a set of at least three points in the plane, and T is the set of all potential lines in a line cover, is 2.*

Proof. First, let $Y = \{p_1, p_2\}$ be the set of any two points in P. This set is shattered by T if every element of 2^Y can be found as the intersection of Y and some element of T. The element $\{p_1, p_2\} \in 2^Y$ is the intersection of Y and the line in T containing the points p_1 and p_2. The elements $\{p_1\}$ and $\{p_2\}$ are each formed as the intersection of Y and a singleton point in T. The element $\emptyset$ is the intersection of Y and any other singleton point (assuming P has at least three points). Thus, the VC-Dimension of (P, T) is at least two.

Next, consider $Y' = \{p_1, p_2, p_3\}$, any set of three points in P. There are two possibilities for this set: Either all three points are collinear or they are not. If all three points are collinear, then there is no element of T containing two of the three points in Y, since all elements of T are either all points along a line or singleton points. Therefore, the set is not shattered by T. If all three points are not collinear, then there is no element of T containing all three points. Thus, the set is not shattered by T. Therefore the largest set shattered by T has size two, and the VC-Dimension of (P, T) is two. □

Lemma 3.4.2 allows us to additionally bound the approximation ratio by $O(\log c)$ where c is the size of the minimum set cover, as described earlier in Section 3.1.2 [26]. The maximum set size corresponds to the maximum number of collinear points, and the minimum set cover corresponds to the minimum number of lines that can cover the points. Thus we have:

Theorem 3.4.3 *Given a set of points P among which no more than z are collinear, and which can be covered by c lines, Algorithm 4 finds an $O(\log(\min(z, c)))$-approximation.*

3.5 An approximation of NC-$\square MLTSP$ using $OPT + 2$ bends

In this section, instances of $\square MLTSP$ where the points are in general position are considered. These instances have the property that no two points share an x-coordinate or a y-coordinate. This formulation is motivated by situations where points are sparsely dispersed over a large area.

In this case, no two points may lie along the same line of the tour, and hence n is a lower bound on the number of bends in the tour. Also the number of lines in any rectilinear tour must be even, since the tour must have the same number of horizontal and vertical lines. Thus if n is odd, then $n + 1$ is a lower bound on the number of lines.

The approximation algorithm described in this section finds a tour with exactly $n + 2$ lines if n is even and exactly $n + 3$ lines if n is odd. Thus, the algorithm finds a tour with at most $OPT + 2$ bends, where OPT is the number of bends in an optimal tour.

3.5.1 Box Points and Diagonal Points

Our algorithm depends heavily on the division of the points into two categories: diagonal points, and box points. Here we define these two sets. In the following section we show that the input to NC-$\square MLTSP$ can always be partitioned into one set of box points and one set of diagonal points.

We say that a set of points $P = \{p_1, p_2, \ldots, p_n\}$ is *monotonically increasing* if, for any two points $p_i = (x_i, y_i)$, and $p_j = (x_j, y_j) \in P$ we have $x_i > x_j$ if and only if $y_i > y_j$. Similarly, P is *monotonically decreasing* if for any two points $p_i, p_j \in P$ we have $x_i > x_j$ if and only if $y_i < y_j$.

Definition 3.5.1 (Diagonal Points) *A set of points may be considered* diagonal points *if*

they can be partitioned into two sets $\mathcal{I}$ and $\mathcal{D}$ such that the points in $\mathcal{I}$ are monotonically increasing and the points in $\mathcal{D}$ are monotonically decreasing.

Definition 3.5.2 (Smallest enclosing rectangle) *Given any set of points, P, define the* smallest enclosing rectangle *to be the rectangular region containing all points of P, bounded by horizontal and vertical lines, having the smallest possible area.*

Definition 3.5.3 (4-box) *Define a* 4-box *to be any set of four points $P = p_1, \ldots, p_4$, such that a single point falls along each of the four boundaries of the smallest enclosing rectangle around P, and no point falls at any corner of that rectangle.*

Definition 3.5.4 (Box Points) *A set of points may be considered* box points *if they can be partitioned into subsets of cardinality 4, such that each subset forms a 4-box, and such that given any two of these 4-boxes, one lies entirely within the other (See Figure 3.4).*

Ultimately we want to show that for any given set of points with the non-collinearity property, all points can be partitioned into a single set of diagonal points and single set of box points as defined above.

3.5.2 Planar Subdivisions

We will classify the input points using a method we call the *Planar Subdivision Method*. We will define two different ways to divide the Euclidean plane, a 4-division and a 9-division. Then we will show that among any set of points with the non-collinearity property there exists a way to divide the plane into a 4-division or a 9-division. See Figure 3.2 for examples of the two divisions.

Definition 3.5.5 (4-division) *A 4-division is a division of the Euclidean plane via a single horizontal and a single vertical line into 4 quadrants, NE, NW, SE, and SW, such that:*

1. *The points in the NW region and the points in the SE region are monotonically decreasing.*

2. *The points in the NE region and the points in the SW region are monotonically increasing.*

Due to the relative positions of the quadrants, it is also the case that the union of the points in the NW and SE regions are monotonically decreasing. Likewise the union of the points in the NE and SW regions are monotonically increasing. In a tour this will allow us to cover these sets of points at a rate of one per line within each set. Note that in order to have a 4-division it is not necessary that a particular quadrant contain any points. Among any arrangement of 0, 1, 2, or 3 points, there exists a 4-division, but it is not not necessarily true that there exists a 4-division among 4 points. For example, there are is not a 4-division of the pointset $\{(-2,-1),(-1,-2),(2,1),(1,2)\}$.

Definition 3.5.6 (9-division) *A 9-division is a partition of the Euclidean plane into 9 regions, NE, NW, SE, SW, N, S, E, W, and C (the Center), defined by two horizontal and two vertical lines, satisfying the properties of the 4-division as well as:*

3. *The N, S, E, and W regions are all empty.*

4. *There exists exactly one point along each of the four boundaries of the center region, and none at any of the corners.*

5. *The interior of the center region may contain any arrangement of points.*

We now define a function PLANAR-SUBDIVISION which, given a set of points with a 9-division, returns NW, NE, SW, SE, C, and B. These are the points in the NW, NE, SW, SE, and C regions respectively, and B is the 4 points along the boundary of C. If no 9-division exists, the algorithm finds a 4-division, and returns NW, NE, SW, and SE, the sets of points in the respective quadrants.

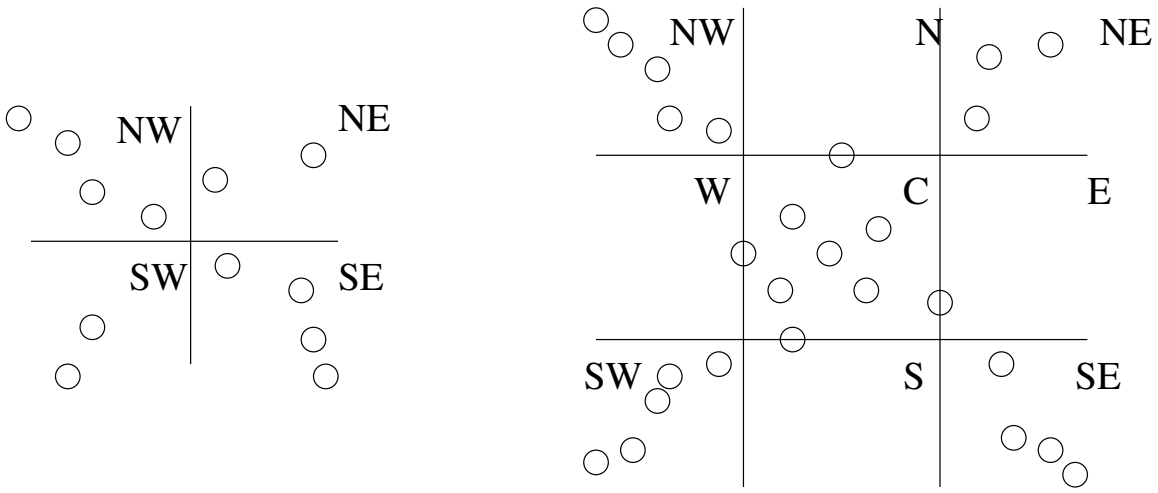


Figure 3.2: A 4-division and a 9-division

The algorithm repeatedly finds the smallest enclosing rectangle around the set of points. If there are 4 points along the border of that box, then a 9-division exists. If there is one point, then a 4-division exists. Otherwise, the box must have a corner point. We remove that corner point, classify it in the appropriate quadrant, and repeat the algorithm on the remaining points. Of the sets returned, B must contain exactly four points, but any of the other sets returned may be empty. Detailed pseudocode appears in Algorithm 5.

In order to analyze the planar subdivision algorithm, several additional properties are needed. Note that if some edge of a smallest enclosing rectangle did not contain any points, then the region would not properly be a smallest rectangle, as we could shrink it on that side. Thus the smallest enclosing rectangle about a set of points with the non-collinearity property must have exactly one point along each of its edges. In the event that fewer than four points fall along the union of all the edges of a smallest enclosing rectangle, then at least one point must lie at a corner of that rectangle. We state this as the corner lemma:

Lemma 3.5.7 (Corner Lemma) *Given a set of points $P = p_1, p_2, \ldots, p_n$ with the non-collinearity property, such that fewer than 4 points lie along the boundary of the smallest enclosing rectangle, R. At least one point in P must lie at a corner of R.*

Algorithm 5 PLANAR-SUBDIVISION(P)

Given: A set of points, P, in the plane for which no two have same x-coordinate or y-coordinate.

Return: A 9-division $\{NW, NE, SW, SE, C, B\}$ or a 4-division $\{NW, NE, SW, SE\}$

1: $NW \leftarrow NE \leftarrow SW \leftarrow SE \leftarrow C \leftarrow B \leftarrow \emptyset$
2: **while** ($P \neq \emptyset$) **do**
3: $R \leftarrow \{maxx, maxy, minx, miny\} \leftarrow$ SMALLEST-ENCLOSING-RECTANGLE(P)
4: **if** ($maxx \neq maxy \neq minx \neq miny$) **then**
5: $DivisionType \leftarrow 9$
6: $C \leftarrow P - R$
7: $B \leftarrow R$
8: **return** $\{DivisionType, NW, NE, SW, SE, C, B\}$
9: **else if** ($maxx = maxy = minx = miny$) **then**
10: $DivisionType \leftarrow 4$
11: $NW \leftarrow NW \cup \{maxx\}$
12: **return** $\{DivisionType, NW, NE, SW, SE, \emptyset, \emptyset\}$
13: **else**
14: **if** ($maxx$= $maxy$) **then**
15: $NE \leftarrow NE \cup maxx$; $P \leftarrow P - maxx$
16: **else if** ($minx$= $maxy$) **then**
17: $NW \leftarrow NW \cup minx$; $P \leftarrow P - minx$
18: **else if** ($maxx$= $miny$) **then**
19: $SE \leftarrow SE \cup maxx$; $P \leftarrow P - maxx$
20: **else if** ($minx$= $miny$) **then**
21: $SW \leftarrow SW \cup minx$; $P \leftarrow P - minx$

Proof. Assume that no point lies at a corner of R. Then there will be at least one side of R along which lie no points. This violates the definition of the smallest enclosing rectangle. □

We now know that given a set of four points P with the non-collinearity property, the following are equivalent:

1. The set P forms a 4-box.
2. The smallest enclosing rectangle about P has four points along its boundary.
3. No point lies at the corner of the boundary of the smallest enclosing rectangle about P.

The 4-box plays an important role in the classification of box points. Here we state the uniqueness of an enclosing 4-box.

Lemma 3.5.8 *Given a set of points $P = \{p_1, p_2, \ldots, p_n\}$ with the non-collinearity property, if there exists a 4-box in P, then there exists a unique 4-box which encloses all other 4-boxes in P.*

Proof. Let R be the smallest enclosing rectangle about all 4-boxes in P. Since no point can lie at the corner of a 4-box, no point can lie at the corner of the enclosing rectangle of a set of 4-boxes. Thus R is an enclosing 4-box. R is also unique. □

Lemma 3.5.9 *Given a set of points $P = \{p_1, p_2, \ldots, p_n\}$ with the non-collinearity property, if there exists a 9-division in P, then* PLANAR-SUBDIVISION(P) *finds the 9-division, and the points returned form the unique enclosing 4-box in P.*

Proof. First, we will show that the algorithm finds the unique enclosing 4-box if it exists. The set P initially contains all points. A point p' is only removed from P when it is on the

corner of the smallest enclosing rectangle around P. Such a point cannot be in a 4-box, since it would have to be at the corner of any 4-box containing p'. Thus, we do not remove any points from P that could be in a 4-box. The algorithm only terminates when all points are removed from P, or when it finds that the smallest enclosing rectangle contains four points along its boundary. Thus it follows that either there does not exist a 4-box in P, or the algorithm finds the unique enclosing 4-box.

Next we show that if there exists a 4-box in P, then the algorithm finds a 9-division. Assume there exists a 4-box in P. Then we know that the algorithm finds the unique enclosing 4-box in P. Assume, by way of contradiction, that this 4-box does not define a 9-division. Then one of the properties of a 9-division would have to be violated. We will attempt to violate each property, and then contradict each violation.

Properties 1 and 2: Let there be two points in a corner region (NW, NE, SW, SE) which do not follow the region's monotonicity property. Then these two points, together with the two far points from the 4-box would form a larger 4-box, and the given 4-box would not be the unique enclosing 4-box.

Property 3: Let there be a point in a side region (N,S,E,W). Then this point together with the 3 far points of the given 4-box would form a larger 4-box.

Properties 4 and 5 hold trivially and thus, our algorithm finds a 9-division if a 4-box exists. Since every 9-division contains a 4-box, our algorithm finds a 9-division if one exists. □

Lemma 3.5.10 *Given a set of points $P = \{p_1, p_2, \ldots, p_n\}$ with the non-collinearity property, if there does not exist a 9-division in P, then there exists a 4-division in P, and* PLANAR-SUBDIVISION *finds a 4-division.*

Proof. First let us show that if there does not exist a 9-division in P, then there exists a 4-division. We do this by induction on the cardinality of a set of points with no 9-division.

Assume there does not exist a 9-division in P and consider the smallest bounding rectangle about P. There cannot be 4 points on this rectangle, or there would be a 4-box and thus a 9-division in P. Therefore, there must be a point, say p_c, at the corner of this rectangle. Now consider the set of points $P - \{p_c\}$. Assume inductively that all sets of $|P| - 1$ points which do not have a 9-division contain a 4-division. Thus $P - \{p_c\}$ contains a 4-division (since it contains no 9-division).

Adding p_c to this set can only eliminate the 4-division if it breaks the monotonicity property in some region. Since p_c was a corner point in the set P, it follows that it is extremal in both the x and y direction, among the points in P. We can show that for at least one region R, p_c will satisfy the monotonicity property for that region.

Thus it maintains a particular monotonicity with every point other point in P, namely the monotonicity of the region which extends infinitely in the same x and y directions. Call that region R. Thus it may only violate the monotonicity policy of the two regions adjoining R. If p_c is located in R as defined on on $P - \{p_c\}$, then the same 4-division as existed in $P - \{p_c\}$ will exist on P.

Consider that p_c is not located in R. Then we can show that it is possible to reassign the boundaries so that p_c is in R without changing the region to which any other point in P is designated.

Then either the N-S boundary or the E-W boundary (or both) must be beyond p_c in its extremal direction. Thus there can be no points between p_c and that boundary. Thus we can reassign that boundary to the other side of p_c without changing the designation of any point in $P - \{p_c\}$. If necessary, we can do this for both boundaries. Thereby, we can change the designation of p_c to R and define a 4-division over P.

Thus, we have our inductive hypothesis. Let P be a set of points of cardinality n which has the non-collinearity property but no 9-division. If there exists a 4-division on any set Q of cardinality $n - 1$ which has no 9-division, then there exists a 4-division on P. For our

base case we simply state that there is a 4-division on any set of points of cardinality 1, and we are done with the inductive proof.

It remains to show that our algorithm finds a 4-division, if there is no 9-division. Assume there is no 9-division on P. Then our algorithm can never find a 4-box in P. Thus at each iteration, it must find a corner point. As this point is extremal in two directions, it maintains a particular monotonicity with all remaining points in P. Thus, assigning it to the region which has that monotonicity property cannot violate the property in that region. Our algorithms does this. Furthermore, the regions remain properly defined, since, when a point is assigned to a region (and removed from P) it is extremal with respect to all remaining points in P in the proper direction. Thus for any two points in different regions, we were guaranteed the proper directional relationship when the first of these was removed from P. Therefore, the 4 regions the algorithm returns will have the proper monotonicity property, and thus they will form a 4-division. □

Theorem 3.5.11 *(Planar Subdivision Theorem) Given any set of points $P = p_1, p_2, ..., p_n$ such that no two points share an x-coordinate or a y-coordinate, there must exist either a 4-division or a 9-division among those points, and* PLANAR-SUBDIVISION(P) *returns a proper 4-division or 9-division among P.*

Proof. This follows from the previous two lemmas. □

3.5.3 The Algorithm

Algorithm 6 describes an approximation algorithm which finds a tour with at most two bends more than that of a minimum link tour. In lines 2-7, it repeatedly applies the planar subdivision theorem to obtain a decomposition of the points. The loop will run for at most $n/4 + 1$ iterations because each iteration, save the last, reduces the cardinality of C by at least 4. Recall that each decomposition partitions the points into sets NW, NE, SW, and

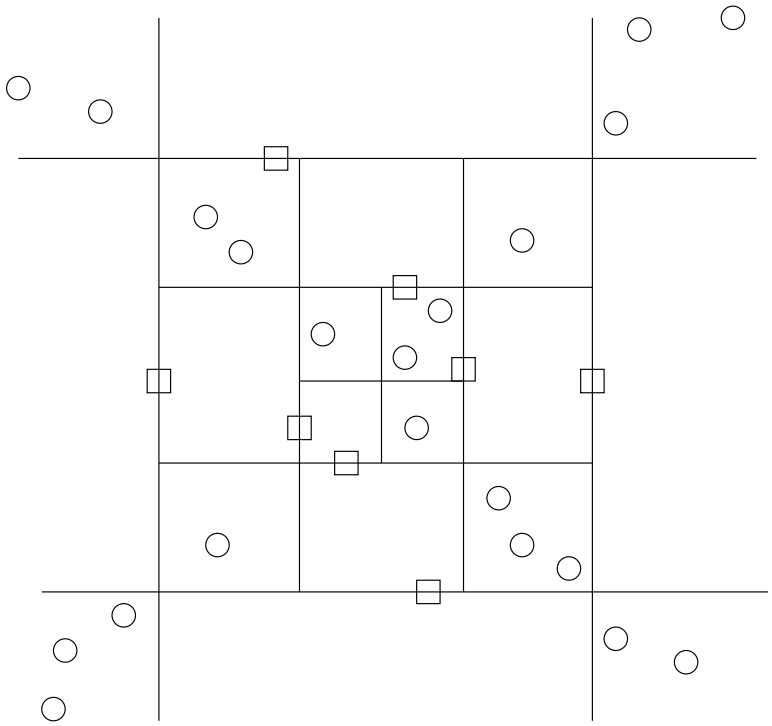


Figure 3.3: Recursively finding a 9-division; finishing with a 4-division at the center. The boxes will become Box Points, and the circles will become Diagonal points.

SE, and in the case of a 9-division, additionally B and C. The points in NW, NE, SW, and SE are placed in the appropriate diagonals, either $\mathcal{I}$ or $\mathcal{D}$, where $\mathcal{I}$ is the union of all points which are in the SW or NE region of a partition, and $\mathcal{D}$ is the union of all points which are in the SE or NW region of a partition. The points in $\mathcal{I}$ and $\mathcal{D}$ are sorted by their y-coordinate. The diagonal $\mathcal{D}$ is further partitioned into its two halves $\mathcal{D} = \mathcal{D}_{SE} \cup \mathcal{D}_{NW}$, where $\mathcal{D}_{NW}$ is the set of all points which are in the NW region of some 9-division, and $\mathcal{D}_{SE}$ is the set of all points which are in the SE region of a 9-division, plus all points within the NW or SE quadrants of the final 4-division. The section $\mathcal{D}_{NW}$ is further partitioned into sets $\mathcal{D}_{NW} = \bigcup \mathcal{D}_i$, where each set is all points contained in the NW region of a single 9-division.

The points which were on the box B of some 9-division are placed into the set Box. These box points $Box = \bigcup Box_i$ are indexed by the iteration in which they were found.

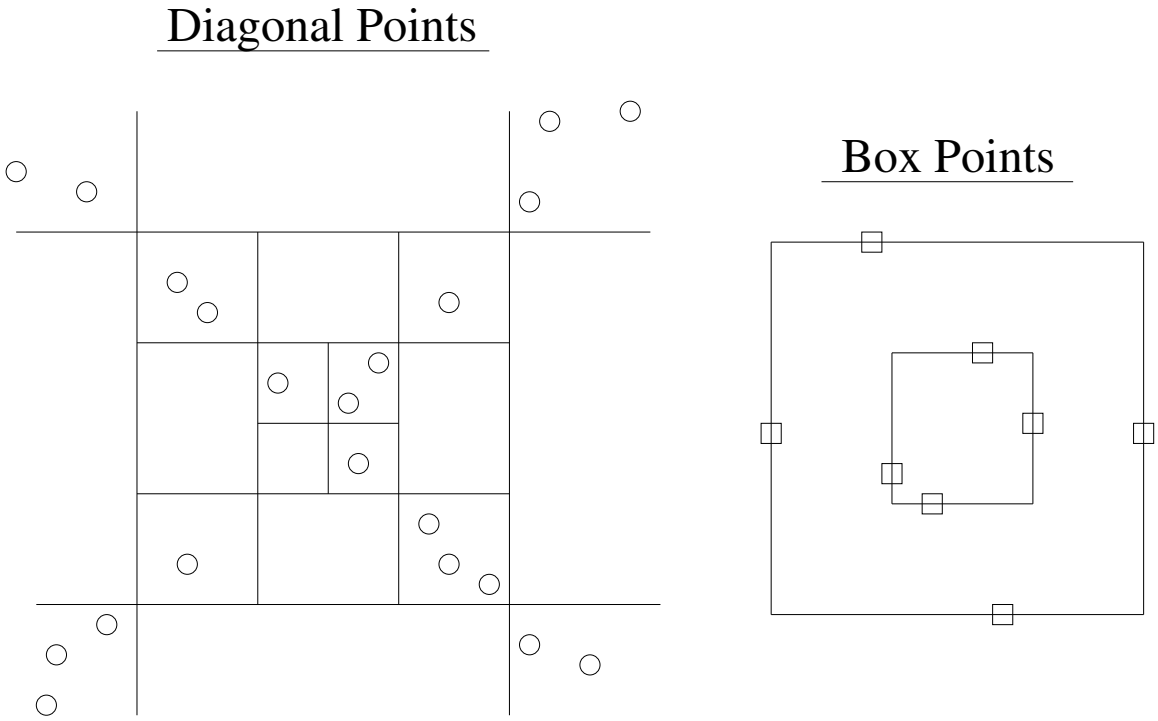


Figure 3.4: Classifying the points into Diagonal Points and Box Points.

The points in any C region are further partitioned into a 4-division or a 9-division.

The tour then consists of the points in $\mathcal{I}$, followed by the points in $\mathcal{D}_{SE}$, and then alternates between the points in $\mathcal{D}_{NW}$ and the various boxes. In order to get the exact bounds claimed, it is necessary to be careful about the starting direction and it may also be necessary to move points from one diagonal to another. The algorithm which performs this switch is called SWITCH-DIAGONAL. These details are discussed in the proofs of Theorems 3.5.12 and 3.5.13.

Theorem 3.5.12 *(Diagonal Switching Theorem) For any set of diagonal points derived using the Planar Subdivision method, there exists at least one point which may be swapped between $\mathcal{D}$ and $\mathcal{I}$, while retaining the monotonicity properties.*

Proof. Recall that we defined the diagonal points as those points in the NE, NW, SE, and SW quadrants. Also recall that those points were relegated to those region by virtue of

being at the corners of a series of rectangles, or being the final point of a 4-division. Note further that for any two of these rectangles, one is completely enclosed within the other.

Consider the innermost rectangle of this series which has a diagonal point at its boundary. If this point is the only diagonal point on this box, then it may be placed in either $\mathcal{I}$ or $\mathcal{D}$ without violating the monotonicity properties. If there are 2 diagonal points on the boundary of this rectangle, then they can both be in the one set from $\mathcal{I}$ and $\mathcal{D}$ whose monotonicity property they share, or they can be split between the two sets $\mathcal{I}$ and $\mathcal{D}$. □

Algorithm 6 FIND-NC-RECTILINEAR-MINLINKTOUR(P)

Given: A set of points, P, in the plane for which no two have same x-coordinate or y-coordinate.

Return: An $OPT + 2$ approximation of a minimum link tour

1: $i \leftarrow 1$
2: **while** ($P \neq \emptyset$) **do**
3: $\{DivisionType, NW, NE, SW, SE, B, C\} \leftarrow$ PLANAR-SUBDIVISION(P)
4: **if** ($DivisionType = 4$) **then**
5: $\mathcal{I} \leftarrow \mathcal{I} \cup NE \cup SW$
6: $\mathcal{D}_{SE} \leftarrow \mathcal{D}_{SE} \cup NW \cup SE$
7: $P \leftarrow \emptyset$
8: **else if** ($DivisionType = 9$) **then**
9: $\mathcal{I} \leftarrow \mathcal{I} \cup NE \cup SW$
10: $\mathcal{D}_{SE} \leftarrow \mathcal{D}_{SE} \cup SE$
11: $\mathcal{D}_i \leftarrow NW$
12: $Box_i \leftarrow B$
13: $P \leftarrow C$
14: $i \leftarrow i + 1$
15: $\mathcal{I}_{Sort} \leftarrow$ SORT$(\mathcal{I})$
16: $\mathcal{D}_{SE-Sort} \leftarrow$ SORT$(\mathcal{D}_{SE})$ // Along the y-coordinate
17: **if** ($|\mathcal{I}_{Sort}|$ is odd) **then**
18: $StartingDirection \leftarrow$ EAST
19: **else if** ($|\mathcal{I}_{Sort}|$ is even) **then**
20: $StartingDirection \leftarrow$ NORTH
21: $\pi \leftarrow \{StartingDirection, \mathcal{I}_{Sort}, \mathcal{D}_{SE-Sort}, Box_{i-1}, \mathcal{D}_{i-1}, \ldots, Box_1, \mathcal{D}_1\}$
22: **if** (FINISHING-DIRECTION$(\pi) = StartingDirection$) **then**
23: **return** SWITCH-DIAGONAL(π,$\mathcal{I}$,$\mathcal{D}_{SE}$) // Apply Diagonal Switching Theorem
24: **else**
25: **return** π

Theorem 3.5.13 *Algorithm* FIND-NC-RECTILINEAR-MINLINKTOUR*, given an input to the Non-Collinear Rectilinear Minimum Bends TSP Problem, finds a tour which contains at most 2 additional bends more than the optimal.*

Proof. As stated earlier, if n is even, then n is a lower bound on the number of lines in the optimal tour, by the non-collinearity property. Similarly, if n is odd, then $n + 1$ is a lower bound. Now consider the number of lines in a tour returned by Algorithm FIND-NC-RECTILINEAR-MINLINKTOUR. The return value consists of a series of point sequences, together with a starting direction. The total number of lines will be the sum of the number of lines in each sequence, plus the sum of the additional lines used in connecting adjoining sequences. An additional line will be necessary between two sequences if the preceding sequence finishes its path heading away from the start of the subsequent sequence. More than one additional line would be necessary only if a particular entrance direction were required at a point. Our definition of a tour defined by points will not require this, except to rejoin the end of a tour to its starting point.

The sorted diagonal point sequences $\mathcal{I}_{Sort} = \text{SORT}(\mathcal{I})$ and $\mathcal{D}_{SE-Sort} = \text{SORT}(\mathcal{D}_{SE})$, because of their monotonicity, can be covered, starting from an extremal point, using a staircase-like path. This can be done at a rate of one line per point, unless the starting direction requires that an extra turn be made in traveling from the first point to the second point. Transferring from $\mathcal{I}_{Sort}$ to $\mathcal{D}_{SE-Sort}$ will incur an extra turn, unless the tour can proceed directly to $\mathcal{D}_{SE-Sort}$, in which case an extra line is needed after the first point of $\mathcal{D}_{SE-Sort}$. Either way the total number of line segments needed to cover $\mathcal{I}_{Sort}$ and $\mathcal{D}_{SE-Sort}$ will be $|\mathcal{I}| + |\mathcal{D}_{SE}| + 1$ (See figure 3.5). Note that this algorithm does not handle the degenerate cases where the sets may have 0 or 1 points, but optimal paths can be found in these instances on a case-by-case basis.

Upon completion of $\mathcal{D}_{SE-Sort}$ the path is inside the innermost uncovered Box_i, heading

in either the North direction or the West direction, and all remaining points in D_i are to the North and West of the most recently covered point. These conditions are invariants to maintain after each subsequent point sequence, D_j, is completed. The conditions are sufficient to cover all points of the inner most uncovered Box_i using exactly 4 additional lines. The invariants remain true upon completion of Box_i. The invariants are also sufficient to cover a particular D_i using $|D_i|$ lines, and remain true after such a covering. Thus all points in $\bigcup_i (\text{Box}_i \cup D_i)$ can be covered with $\sum_i(|\text{Box}_i| + |D_i|)$ lines (See figure 3.6).

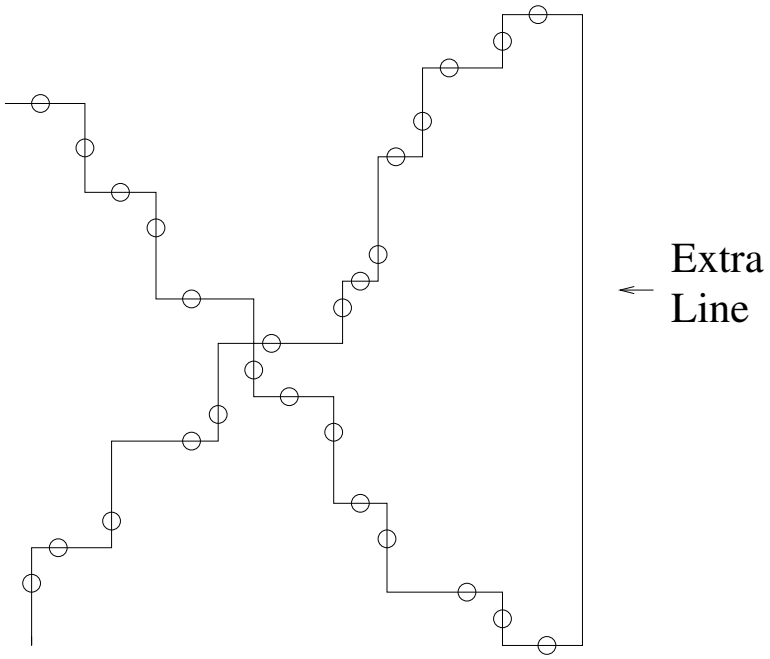


Figure 3.5: Joining the two diagonal paths

Joining the end of the tour back to its beginning will require 1 or 2 or 3 additional lines, depending on the starting and finishing directions. If 3 additional lines are required, then the starting and finishing directions must both be North. In this case we can apply the Diagonal Switching Theorem to modify the sizes of $\mathcal{D}_{SE}$ and $\mathcal{I}$ each by 1. This changes both the starting and finishing directions, allowing the tour to complete with 1 additional line. The resultant tour will have $n + 2$ total lines, or $n + 3$ total lines. By the parity

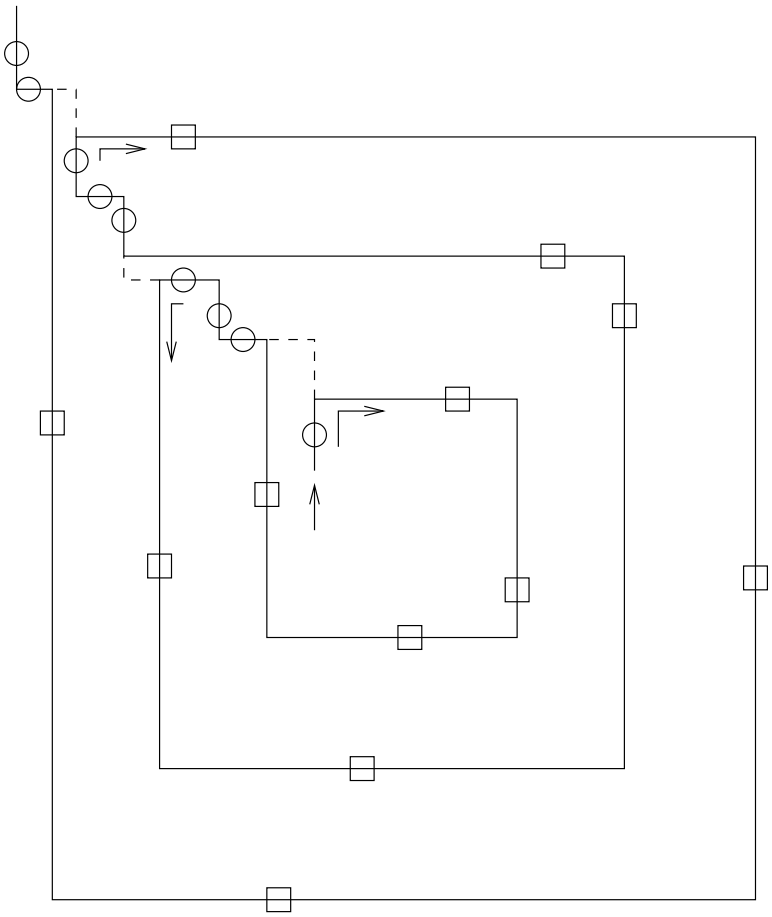


Figure 3.6: Splicing in the box points

argument, the tour can only have $n + 3$ lines if n is odd. Thus we achieve the desired optimality bound. □

The total number of iterations of both the *while* loops in FIND-NC-RECTILINEAR-MINLINKTOUR and in PLANAR-SUBDIVISION is at most n, since an iteration of either loop reduces the size of P. The SMALLEST-ENCLOSING-RECTANGLE is thus run at most n times, and if run in a brute force fashion takes $O(n)$ time to run.

However, each successive call to SMALLEST-ENCLOSING-RECTANGLE returns a smaller rectangle. Thus by first sorting the points along both axes, the smallest enclosing rectangle can be found by stepping through the points in order, beginning where the last call left

off. The resulting sequence of calls takes $O(n \log n)$ overall, including the time to sort the points.

The remainder of the algorithm runs in $O(n)$ time.

3.6 An NP-Completeness Proof

The Minimum Link Traveling Salesman Problem (MLTSP) was introduced in Section 3.4. In that section, we have seen an approximation ratio for this problem of $O(\log(\min(z, c)))$, where z is the maximum number of collinear points, and c is the size of the minimum line cover. This ratio can be obtained by representing the problem as a Set Cover problem, and using known algorithms for Set Cover. Here the NP-Completeness of the general problem is shown.

Author's Note

After the original writing of this section we discovered that the main theorem described herein had been proven independently by Arkin, Mitchell, and Piatko in an unpublished report. That report has since been published, and now appears as Lemma 1 of [11]. Their methodology is similar to that described here, involving a reduction from the same problem, and is somewhat easier to describe. It is reviewed in Section 3.6.5 for the benefit of comparing the two proofs.

3.6.1 Reduction

Theorem 3.6.1 *The decision version of the Minimum Link Traveling Salesman Problem is NP-Complete, by a reduction from Line Cover.*

1. To see that the decision problem is in NP, note that the sequence of line segments forms a certificate of polynomial size.

2. Next reduce the general Line Cover Problem (see below) to MLTSP. Given a Line Cover instance which contains n points, construct an instance of MLTSP containing $n + 6mn + 36m^2$ points as described in section 3.6.2, where m is the number of unique lines that pass through two or more points of the Line Cover instance. In section 3.6.3 it is proven that the MLTSP instance can be covered with a tour of $k + 6m$ lines if and only if the original set of points has a line cover of size k.

Line Cover

Recall from Section 3.1.4 that the Line Cover problem asks for the minimum number of lines which may cover a set of points in the Euclidean plane. The problem was proven NP-Complete by Megiddo and Tamir in 1982 [116].

3.6.2 Construction

Form a reduction from Line Cover to MLTSP by constructing an instance of MLTSP which will solve a given Line Cover problem. Let $P = \{p_1, p_2, \ldots, p_n\}$ be the set of points which comprise an instance of Line Cover. Formulate P', an instance of MLTSP to consist of the union of the points in P, and another set of points, Q, which we define in this section.

Define $L = \{l_0, l_1, \ldots, l_{m-1}\}$ be the set of all unique lines which pass through at least two distinct points in P. We note that $m \leq \binom{n}{2}$.

Let C be the smallest circle which falls outside all intersections between lines in L. Note that the number of intersections is finite (at most $\binom{m}{2}$), so some C must exist. Define c to be the center of C. Define another circle, C', centered at c, whose radius is three times that of C.

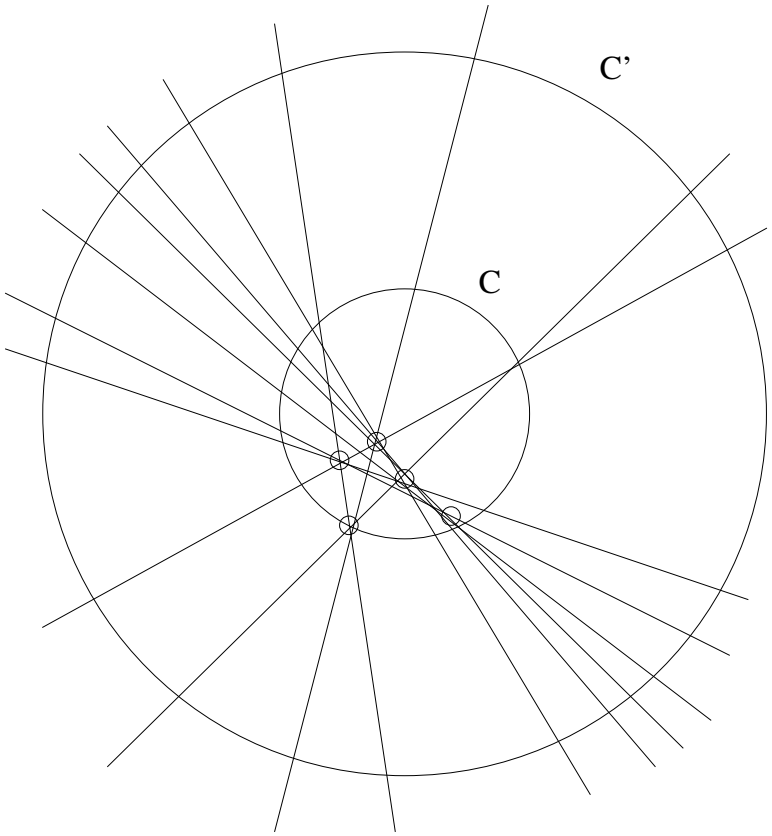


Figure 3.7: The Circles C and C'

Consider the set of $2m$ points defined by the intersections between C' and the lines in L. Call this set $Q_{C'}$ Note that there are exactly $2m$ such points, and each is unique.

Select a point $q_0 \in Q_{C'}$ as follows. Consider the topmost point $q_T \in Q_{C'}$ as a candidate for q_0. Let $q_{T-1} \in Q_{C'}$ be the next point appearing in a counterclockwise direction around C'. Let $l_T \in L$ be the line passing through q_T and let $l_{T-1} \in L$ be the line passing through q_{T-1}. If $l_T \neq l_{T-1}$ and l_T is not parallel to l_{T-1}, then accept $q_0 \leftarrow q_T$. Otherwise, if either of these conditions hold, reject q_T and consider the next point $q_{T+1} \in Q_{C'}$ appearing in a clockwise direction around C' as a candidate for q_0. Let $l_{T+1} \in L$ be the line passing through q_{T+1}. If $l_{T+1} \neq l_T$ and l_{T+1} is not parallel to l_T then accept $q_0 \leftarrow q_{T+1}$. Otherwise consider the next point $q_{T+2} \in Q_{C'}$ appearing in a clockwise direction around C' as the next candidate. Continue to consider candidates in this way, until an acceptable q_0 is found.

Note that unless $m = 1$ (which can only happen when all the points of P are collinear), such a q_0 must exist.

Label the points $Q_{C'} = \{q_0, q_1, \ldots, q_{2m-1}\}$, beginning at q_0 of the circle, and proceeding in a clockwise fashion around C'.

Theorem 3.6.2 *Each of* $\{q_x : 0 \leq x < m\}$ *falls along a unique line in* L.

Proof. Assume there exists a line $l_a \in L$ passing through two points among $[q_0, q_{m-1}]$. Since there are m lines there must then exist another line l_b with two of its endpoints among $[q_m, q_{2m-1}]$. By the ordering of the endpoints, l_a and l_b cannot intersect within C'. Neither can they intersect outside of C' so they must be parallel. By delaying the choice of q_0 we guaranteed that q_0 and q_{2m-1} do not lie along parallel lines. So l_a and l_b cannot be the two lines passing through these two points. Therefore, there must be some line, $l_c \in L$, passing through at least one of these two points which is not parallel to both l_a and l_b. So l_c intersects both l_a and l_b, but it exits C' in between l_a and l_b. This implies that l_c intersects one of either l_a or l_b outside of C', which is a contradiction since we defined C to enclose all intersections. □

Now, without loss of generality, define l_x to be the line which q_x falls along for all $x < m$. This gives us an ordering of the members of L.

We will define three line segments, l_{xL}, l_{xR} and l_{xC}, associated with each point q_x. Let q_y be the subsequent point ordered clockwise around around C'. That is, $y = (x + 1) \bmod 2m$.

Define l_{xL} and l_{xR} to begin at q_x. If a and b are points along l_{xL} and l_{xR} respectively, then we define the angles $\angle cq_xa$ and $\angle cq_xb$ to be $5\pi/6$. The former is a left-hand turn, while the latter is a right-hand turn (See figure 3.8).

If the lines defined by l_{xR} and l_{yL} intersect, then let them each terminate halfway between their respective starting points and their point of intersection. If the angle $\angle q_xcq_y$ is

at least $\pi/3$ then these lines will not intersect. If this is the case, let them terminate after a distance equal to twice the radius of the outer circle, C'. Define l_{xC} to be the line segment between the termination points of l_{xR} and l_{yL}. Note that $\angle q_x c q_y$ is at most π (occurring when $m = 1$), and so this means we have defined l_{xR} and l_{yL} long enough so that l_{xC} does not enter C'.

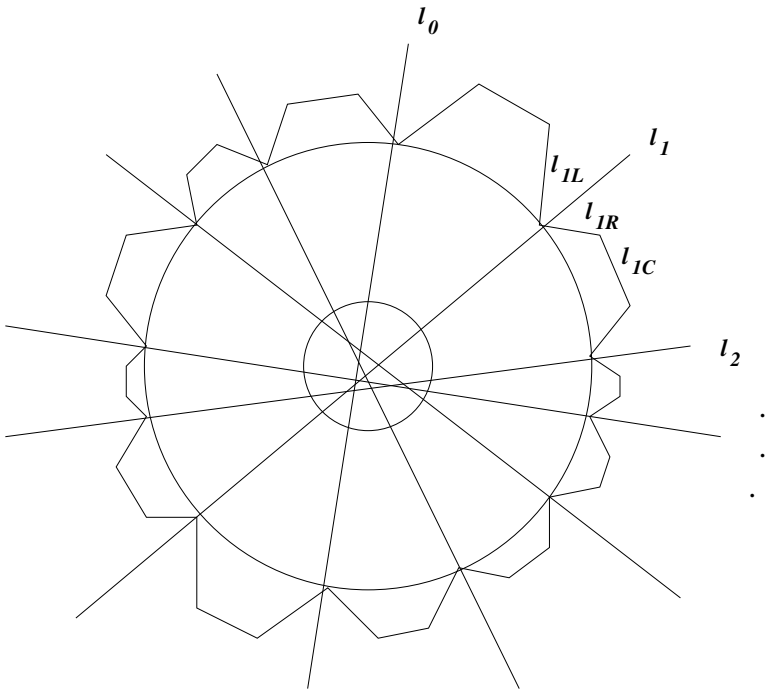


Figure 3.8: The lines l_x, l_{xL}, l_{xC}, and l_{xR}

We now will define Q to consist of points which lie along the line segments which we defined in this section.

For each $x < 2m$ let $\{q_{x0}, q_{x1}, \ldots, q_{x(D-1)}\}$ be a set of $D = n + 6m$ points which lie along l_{xL}. Define these points so as to be unique and not to be collinear with any two other points in Q, save those along l_{xL}. Also let $\{q_{xD}, q_{x(D+1)}, \ldots, q_{x(2D-1)}\}$ be a set of D points which lie along l_{xR}, and $\{q_{x(2D)}, q_{x(2D+1)}, \ldots, q_{x(3D-1)}\}$ be a set of D points which lie along l_{xC}. Define these similarly so as to be unique and not to be collinear with any two other points in Q. Since $D = n + 6m$, this gives us a total of $6mD = 6mn + 36m^2$ points

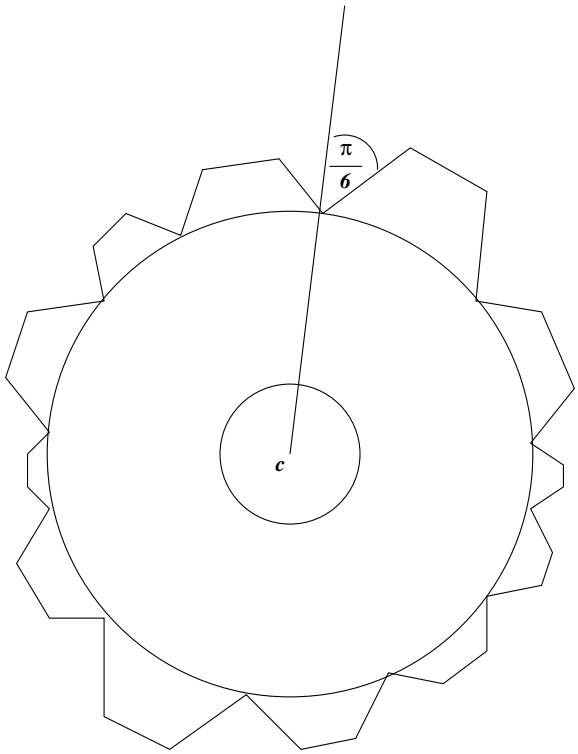


Figure 3.9: The angle of l_{xL}

in Q.

Thus Q is defined as follows:

$$Q = \{q_{xy} : x < 2m, y < 3D\}$$

Let P' be defined as the union of P and Q:

$$P' = P \bigcup Q$$

Our new problem is formulated as follows. Given the set of points P', does there exist a tour covering P' with $k + 6m$ lines?

We show that this is true iff there exists a line cover over P consisting of k lines.

Properties

Consider the line defined by l_{xR} for some x. This line intersects C' at q_x and some other point. Label that other point q_{xR}.

Property 3.6.3 $\angle q_x c q_{xR} = \frac{2\pi}{3}$ *(since* $\angle c q_x a = \frac{5\pi}{6}$ *implies* $\angle c q_{xR} q_x = \angle c q_x q_{xR} = \frac{\pi}{6}$*.) We will take advantage of this fact later on in our tour formulation.*

Property 3.6.4 *The line segment between* q_x *and* q_{xR} *will not enter* C*. (This can be seen by noting that an equilateral triangle can be inscribed between the two circles, and that* $\overline{q_x q_{xR}}$ *is the edge of one such triangle.)*

3.6.3 Proof

Lemma 3.6.5 *Given a set of points* P*, define* P' *as above. There exists a tour covering* P' *with k+6m lines iff there exists a line cover over* P *of size k.*

A Line Cover implies a Tour

Assume there exists a Line Cover J over P consisting of k lines. Note that $J \subseteq L$.

Consider the $2k$ points defined by the intersections between C' and the elements of J. Label those points $S = \{s_0, s_1, \ldots, s_{2k-1}\}$ ordered in a clockwise fashion around C'. Select s_0 so that the line in J passing through s_0 is not parallel to the line passing through s_{2k-1}. Define $j_x \in J$ to be the line which s_x falls along for all $x < k$. Also define s_{xR} and s_{xL} to be the line segments l_{yR} and l_{yL} for which $j_x = l_y$.

Define an angle ϕ_x associated with each line j_x to be the angle $0 \leq \phi_x < \pi$ by which a vertical line needs to be rotated clockwise in order to be parallel with j_x.

We formulate a tour over P' of size $k + 6m$ as follows:

Begin with a tour of $6m$ lines covering the points of Q in a straightforward clockwise order. That is, our initial tour is $\Pi = \{t_0, t_1, \ldots, t_{6m-1}\}$ where t_{3x} is l_{xL}, t_{3x+1} is l_{xR}, and t_{3x+2} is l_{xC}.

Next we will intersperse the tour with the k line segments which make up J, the line cover of P. Thereby shall we have a tour of size $6m + k$.

We will add these k lines into the tour in matched pairs. We will use one of the following two matchings:

1. $\{\{j_0, j_1\}\{j_2, j_3\}...\{j_{k-2}, j_{k-1}\}\}$
2. $\{\{j_1, j_2\}\{j_3, j_4\}...\{j_{k-1}, j_0\}\}$

Our choice between these two matchings is motivated by a desire to keep the differences between the slopes of matched lines small. Thus if j_x is matched to j_y (with $y = (x + 1) \bmod k$) we will require that $(\phi_y - \phi_x) \bmod \pi \leq \pi/2$. Since $0 \leq \phi_x < \pi$ for any x, there can be at most one value for x which violates this constraint. Thus we shall chose the matching which avoids this violation.

This allows us to take advantage of the following theorem:

Theorem 3.6.6 *Let j_x and j_y be two non-parallel lines in J and let i be their point of intersection. If $\angle s_x i s_y \leq \frac{\pi}{2}$ then $\angle s_x c s_y \leq \frac{3\pi}{2} - 2\arccos(\frac{1}{3\sqrt{2}}) < \frac{2\pi}{3}$.*

Proof. The theorem can be proven by a geometric argument. The greatest difference between $\angle s_x i s_y$ and $\angle s_x c s_y$ is achieved when j_x and j_y intersect at the boundary of C, and when c bisects $\angle s_x i s_y$. See Figure 3.10.

Let i fall along the border of C and let $\angle s_x i s_y = \pi/2$ with c bisecting it. Note that if i is moved closer to c while maintaining $\angle s_x i s_y = \pi/2$ then $\angle s_x c s_y$ will decrease. Also note that if s_x and s_y are moved along the border of C' while maintaining $\angle s_x i s_y \leq \pi/2$ then $\angle s_x c s_y$ will decrease.

Now let a be a point along $\overline{is_x}$ such that $\angle cai$ is a right angle. If the radius of C is 1, then it follows that the length of $\overline{ia}$ is $\frac{1}{\sqrt{2}}$, the radius of C' is 3, and $\angle acs_x = \arccos(\frac{1}{3\sqrt{2}})$. This then implies that $\angle s_x c s_y = 2\pi - 2\arccos(\frac{1}{3\sqrt{2}}) - \frac{\pi}{2}$.

□

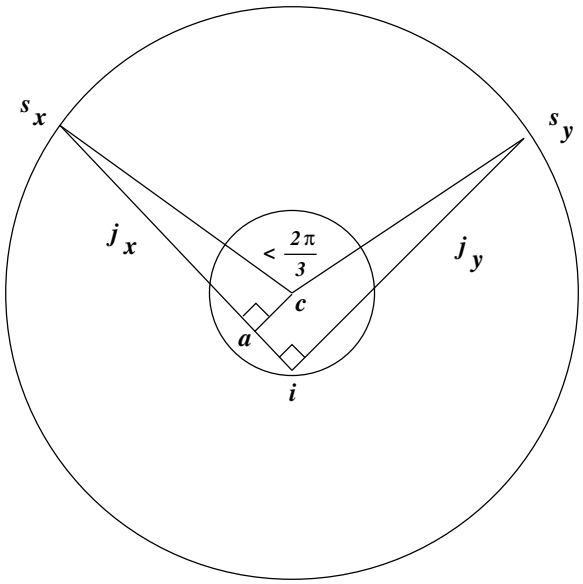


Figure 3.10: An upper bound on the size of the angle $s_x c s_y$

Theorem 3.6.7 *Let j_x and j_y be two parallel lines in J. It follows that $\angle s_x c s_y \leq \pi - 2\arccos(\frac{1}{3}) < \frac{2\pi}{3}$.*

Proof. Consider two parallel lines j_x and j_y which are tangent to opposite sides of C. If the radius of C is 1, then the line $\overline{s_x s_y}$ has length 2, and the lines $\overline{cs_x}$ and $\overline{cs_y}$ each have length 3. Now consider the triangle $\triangle s_x c s_y$. Each of $\angle s_y s_x c$ and $\angle s_x s_y c$ is of size $\arccos(\frac{1}{3})$. Thus it follows that $\angle s_x c s_y$ is of size $\pi - 2\arccos(\frac{1}{3})$. If either of j_x or j_y is moved closer to the other, then $\angle s_x c s_y$ will be smaller. □

These then give us the following important corollaries.

Corollary 3.6.8 *Let j_x and j_y be matched. It follows that $\angle s_x c s_y < \frac{2\pi}{3}$.*

Corollary 3.6.9 *Let j_x and j_y be matched (with $y = (x+1) \bmod k$). It follows from Property 3.6.3 that the lines defined by j_x and j_{yR} intersect within C'.*

Corollary 3.6.10 *Let j_x and j_y be matched (with $y = (x+1) \bmod k$). It follows from Property 3.6.3 that the lines defined by j_{xL} and j_{yR} intersect within C'.*

Now we will describe how to include the lines from J into our tour. As we turn these lines into line segments of the tour, it is important that the line segments go all the way through the circle C, so that any points within C are still covered. Also it is important that we include them without adding any extra lines to the tour, beyond adding one for every line in J.

Let j_x and j_y be two matched lines in J with $y = (x+1) \bmod m$. We are going to splice these two lines into the tour. Let s_w lie along j_x at the intersection with C' opposite s_x, and let s_z lie along j_y at the intersection with C' opposite s_y. If j_x and j_y are parallel, then instead let s_w and s_z lie along j_y and j_x respectively.

Our original tour appears in the following sequence:

$$\Pi = \{l_{0L}, ..., s_{xL}, s_{xR}, ..., s_{yL}, s_{yR}, ..., s_{wL}, s_{wR}, ..., s_{zL}, s_{zR}, ...l_{(2m)C}\}$$

If j_x and j_y are not parallel, then inserting them into the tour will cause the sequence to become:

$$\Pi = \{l_{0L}, ..., s_{xL}, s_{xR}, ..., s_{yL}, j_y, s_{zL}, ..., s_{wR}, j_x, s_{yR}, ..., s_{wL}, s_{zR}, ..., l_{(2m)C}\}$$

If j_x and j_y are parallel, then inserting them into the tour will cause the sequence to become:

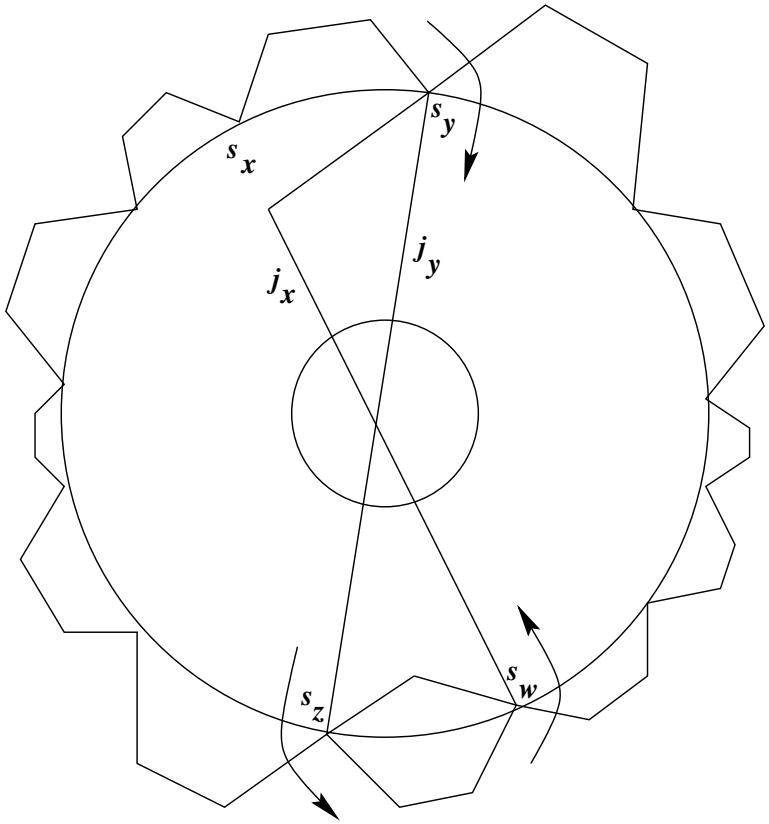


Figure 3.11: The tour, after splicing in non-parallel lines j_x and j_y

$$\Pi = \{l_{0L}, ..., s_{xL}, s_{xR}, ..., s_{yL}, j_y, s_{wR}, ..., s_{zL}, j_x, s_{yR}, ..., s_{wL}, s_{zR}, ..., l_{(2m)C}\}$$

See figures 3.11 and 3.12.

In order to verify that our new tour is well formed, we must ensure that consecutive lines intersect in the appropriate place. Here this means they should intersect within C' or at the border of C', and outside of C. If j_x and j_y are not parallel, it is straightforward to see that j_x intersects with s_{wR}, and that j_y intersects with both s_{yL} and s_{zL} at the border of C'. If j_x and j_y are parallel, then we can see that j_x intersects with s_{zL}, and that j_y intersects with both s_{yL} and s_{wR} at the border of C'. However, we still need to verify that:

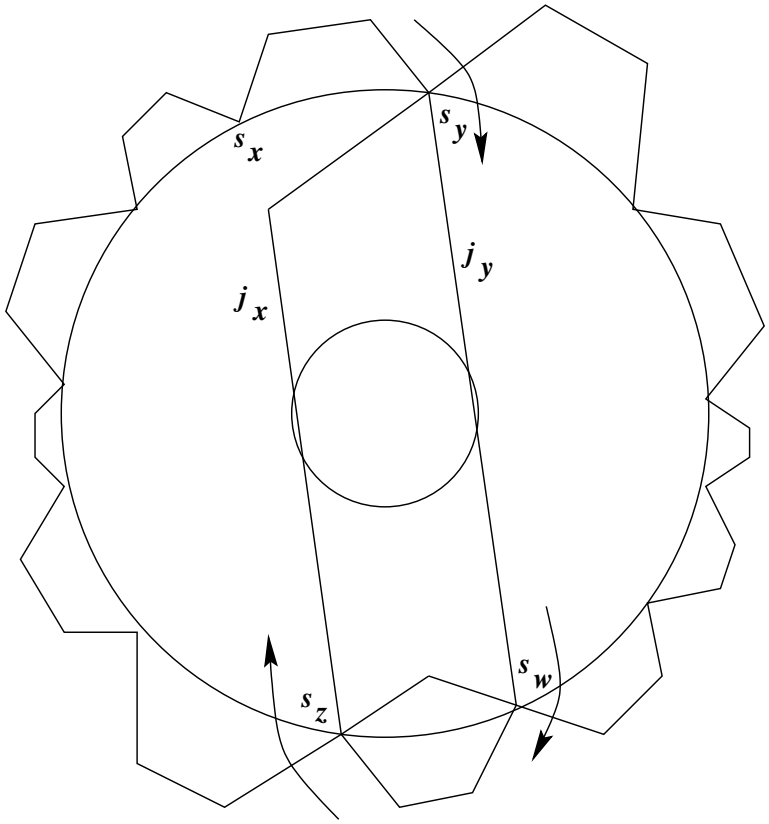


Figure 3.12: The tour, after splicing in parallel lines j_x and j_y

1. j_x intersects with s_{yR} within C' and outside of C

2. s_{wL} intersects with s_{zR} within C'

We know from Property 3.6.4 that these intersections must take place outside of C. We can also show that these intersections happen within C'. In turn each of them follows from:

1. Corollary 3.6.9

2. Corollary 3.6.10

Thus, we are able to add these lines from the line cover into the tour without adding any extra lines to the tour, and while still covering the points in P.

Now that we have shown how to insert the first pair of lines into the tour, we need to show that we can insert the remaining lines. We will do this using induction.

Consider a single instance of splicing a pair of non-parallel matched lines, j_x and j_y (with $y = (x + 1) \bmod k$), into the tour. Let s_w fall along j_x opposite s_x, and let s_z fall along j_y opposite s_y. This splicing will consist of modifying two sections of the tour:

1. ..., s_{yL}, s_{yR}, ... becomes

   ..., $s_{yL}, j_y, s_{zL}, ..., s_{wR}, j_x, s_{yR}$, ...

2. ..., $s_{wL}, s_{wR}, ..., s_{zL}, s_{zR}$, ... becomes

   ..., s_{wL}, s_{zR}, ...

Note that the splicing does not involve any modifications outside of these two sections.

Now assume we have successfully spliced the first i pairs of lines in order of matched pairs into the tour. We wish to splice in the $(i + 1)$th pair of lines. Let j_{xi} and j_{yi} be the ith pair spliced.

Consider that we are trying to splice in $j_{x(i+1)}$ and $j_{y(i+1)}$. This means we will modify the sections ..., $s_{y(i+1)L}, s_{y(i+1)R}$, ... and ..., $s_{w(i+1)L}, ..., s_{z(i+1)R}$, ... of the tour. We need to show we can do this, without disrupting any previous modifications.

The modification of the section ..., $s_{y(i+1)L}, s_{y(i+1)R}$, ... occurs clockwise around the circle later than section $s_{y1L}, ..., s_{yiR}$ since since $s_{y(i+1)}$ is clockwise around the circle later than any of $s_{y1}, ..., s_{yi}$. Also this section appears clockwise before the section $s_{w1L}, ..., s_{ziR}$ since s_{wj} or s_{zj} appears after s_{yk} for any j, k. Thus it follows that modification of this section cannot interfere with any of the previous modifications.

The modification of the section ..., $s_{w(i+1)L}, ..., s_{z(i+1)R}$, ... occurs clockwise around the circle later than section $s_{w1L}, ..., s_{ziR}$ since since $s_{w(i+1)}$ and $s_{z(i+1)}$ are clockwise around the circle later than any of $s_{w1}, ..., s_{zi}$. Also this section appears clockwise before the

section $s_{y1L}, ..., s_{yiR}$ since s_{yj} appears after s_{wk} or s_{zk} for any j, k. Thus it follows that modification of this section cannot interfere with any of the previous modifications.

Therefore, we know by induction that we can splice in all $\frac{k}{2}$ pairs of matched lines into the tour, while only adding k lines. Thus we have shown how to find a tour covering P' using $k + 6m$ lines (bends).

A Tour Implies a Line Cover

Next we need to show that if a there exists a tour of length $k + 6m$ covering P' then there exists a line cover of size k covering P.

Assume there exists a tour of length $k + 6m$ covering P'. Now consider the possibility that such a tour does not contain a line segment which is a subset of the line defined by l_{xR} for some $x < 2m$. Then there must be D distinct line segments covering the points along q_{x0}, q_{x1}, ..., $q_{x(D-1)}$. We defined $D = n + 6m$ at the time of construction. Since $n > k$, we have more than $k + 6m$ lines in the tour, a contradiction. Thus the tour must contain some line segment along each line defined by l_{xR}, and for the same reason along each line defined by l_{xC}, and l_{xL} for all $x < 2m$. This is a total of $6m$ line segments.

Since none of these line segments can cover any points in P, there remain just k lines in the tour to cover all the points of P. These k lines form a line cover over P of size k.

Thus we have shown that if there exists a tour covering P' with $k+6m$ lines, then there exists a line cover covering P with k lines.

3.6.4 Conclusions

Thus it has been shown how to do the following. Given a set of n points which comprise an instance of Line Cover, an instance of MLTSP containing $n + 6mn + 36m^2$ points can be constructed, where m is the number of unique lines which go through two or more points

in the instance of Line Cover. Note that $m \leq \binom{n}{2}$.

This construction has the property that, if the Line Cover instance can be covered with k lines, then the MLTSP instance can be covered with $k + 6m$ bends, and if the MLTSP instance can be covered $k + 6m$ bends, then the Line Cover instance can be covered with k lines. This is sufficient to form a reduction from Line Cover to MLTSP, and proves the latter NP-Complete.

3.6.5 Another NP-Completeness Proof

Here we briefly outline the NP-Completeness proof given by Arkin, Mitchell, and Piatko in 2003 for the purpose of comparing the two proofs [11]. Initially, this proof follows a similar methodology to that described above, including a reduction from Line Cover. It differs, however, in the manner in which additional lines are forced after the reduction.

Begin again with a set of points P which constitute an instance of the Line Cover Problem. Reduce this to the Minimum Link Tour Problem as follows.

As before, consider the arrangement L of all lines going through two or more points of P. Place a series of r "V" shaped wedges, each parallel to the others, and each enclosing the entire point set P, where r is the size of the line cover that is being tested for existence. Using an affine transformation over P if necessary, ensure that every line in L passes through both sides of every wedge. See Figure 3.13.

Along both boundaries of every wedge, place enough additional points to guarantee that any tour must follow both boundaries of every wedge. Place all of these points close enough to the apex of the "V" so that a they all fall in between the same pair of lines in L. Call these points Q.

Theorem 3.6.11 *There exists a line cover over P consisting of r lines if and only if there exists a tour of the points in $P \cup Q$ of size $3r$.*

Proof. If there exists a line cover over P consisting of r lines then a tour having $3r$ links can be constructed as follows. Alternate between following the section of a line in the line cover in between the two sides of the set of wedges, and the two edges of a "V". There will be r links following the line cover, and $2r$ links following the boundaries of the wedges, for a total of $3r$ links in the tour.

Next any tour over this construction must include the $2r$ lines covering the set of wedges, since enough points have been placed here to guarantee this. Thus if there exists a tour of $P \cup Q$ of size $3r$, then there can only be at most r lines used to cover the pointset P. Therefore, there must exist a line cover over P of size r. □

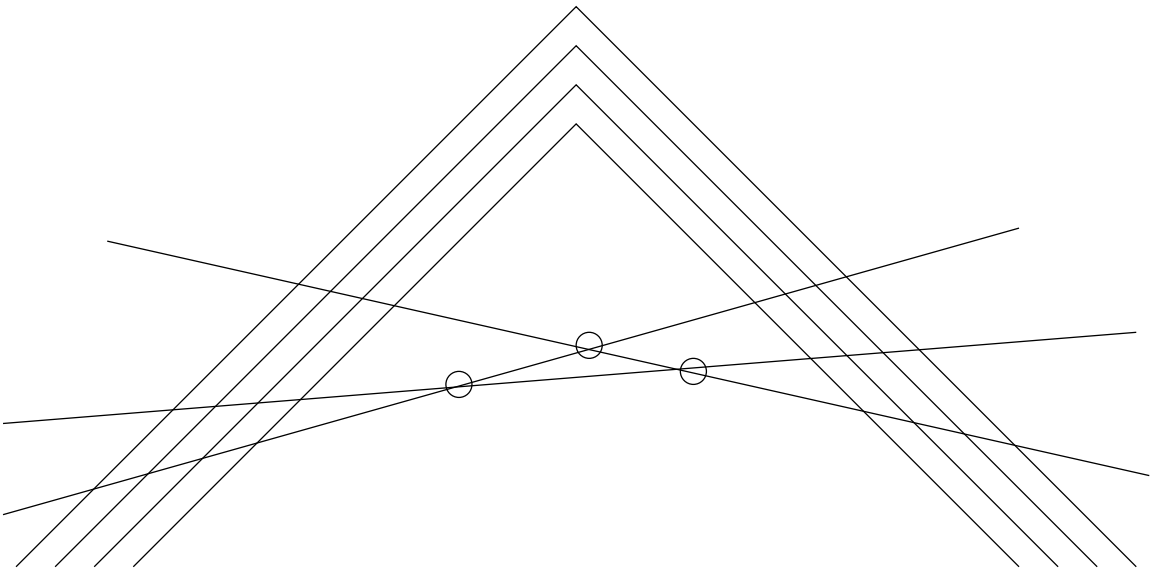


Figure 3.13: A construction of the Arkin-Mitchell-Piatko reduction

Chapter 4

The Minimum Link Path Problem

The *Shortest Path Problem* attempts to find a path through a graph between two vertices such that the total weight of all edges along the path is minimized. This is a fundamental problem in the field of Graph Theory. Dijkstra's algorithm is the widely accepted standard method for solving the problem [39], although its variants have inspired much further discussion. See, for example, the 1973 Ph.D. thesis of Johnson [77]. Its geometric counterpart has also received considerable attention, and has been the central subject in a number of surveys and chapters of of books [120, 122], as well as Ph.D. theses, such as that of Mitchell from 1986 [118].

The Minimum Link Path Problem (sometimes referred to as the Minimum Bends Path Problem) attempts to find a path between two points which has a minimum number of turns. Equivalently, one can minimize the number of straight line segments, or "links", in the path. An entire book chapter by Maheshwari, Sack, and Djidjev is devoted to the subject of minimum link path problems [112]. Rectilinear versions of this problem received considerable attention during the early and mid 1990's, notably including [123, 95, 94, 104, 22]. Most of these treatments have been confined to just two dimensions, since much of the interest in this problem has been motivated by applications in VLSI [94].

A notable exception comes from deBerg, et al. who have considered rectilinear variants of this problem in d dimensions [22]. They describe an $O(n^d \log n)$ method to solve a combined metric problem where the objective is to minimize the total distance traveled, plus some constant C times the total number of bends in the path. Setting C to zero or to a sufficiently high number solves the Euclidean Shortest Path Problem, or the Minimum Link-Distance Problem, respectively.

Here we focus on the Minimum Link-Distance problem with rectilinear (axis-parallel) paths among rectilinear obstacles (with axis-perpendicular faces) in three dimensions. This problem is motivated by applications where moving in a straight line incurs a similar cost, regardless of the distance traveled (for example, in space travel), or where turning is a relatively expensive operation (for example, when using heavy machinery which is difficult to turn). A motivation for our consideration of this problem has been its application to Homogeneous Modular Robots, as described by Fitch, Butler, and Rus in [54].

In this chapter an algorithm is given which improves on the previously best known bound of $O(n^3)$ (see [54] and Section 4.2.1), and runs in $O(\beta n \log n)$ time, where β is the size of a BSP decomposition of the space containing the obstacles. It has been shown that in the worst case $\beta = \Theta(n^{3/2})$ [138], however in many practical circumstances, $\beta \approx \Theta(n)$.

4.1 Background and Related Problems

The algorithm described in this chapter relies on a number of outside results.

Perhaps the first question to consider is deciding upon the best obstacle representation to be used. In three dimensions it is not entirely obvious how best to represent a polyhedron, even if the polyhedron is rectilinear. This matter is discussed in Section 4.1.1.

Next, the algorithm here operates on a set of rectangular regions in three dimensional space. Therefore, once an obstacle representation is chosen, it is necessary to convert these

obstacles into an acceptable input for the algorithm. This happens in two stages.

The first of these stages is to partition each of the rectilinear faces of the obstacle representation into rectangles. In the second stage, the set of rectangles is used to define a *Binary Space Partition (BSP)* of the space containing the obstacles. Polygon partitioning is discussed in Section 4.1.3 and binary space partitioning is discussed in Section 4.4.2.

4.1.1 Obstacle Representations

Historically, various data structures have been used to represent three dimensional polyhedrons.

One of the earlier and most popular is Baumgart's *winged-edge* representation, which was described in 1975 [17]. Under this model, there is a separate data structure for each edge, vertex, and face, and each of these is kept in a complete list of all items of the same type. Every edge has a pointer to its two incident faces and to the four adjacent edges. Note that all edges of a rectilinear polyhedron must have exactly four adjacent edges (two at each end), assuming polyhedrons which touch at a vertex or edge are not considered adjacent. Additionally, every face has a pointer to an arbitrary edge surrounding it, and every vertex has a pointer to an arbitrary edge of which it is an endpoint.

Note that this data structure cannot represent polygonal faces with holes, even if the polyhedron has no holes, since each face only has a single pointer to an edge. This can be easily fixed though, at the loss of constant-sized faces, by allowing a face to have multiple pointers, one to an edge of each hole and one to an outer edge.

Another more flexible representation, known as the *quad-edge* data structure, was described by Guibas and Stolfi in 1985 [64]. This again makes the edge the primary data structure. Each edge keeps pointers to four edges, one to the next edge moving around each of its two incident vertices in a counterclockwise direction, and one to the next edge

moving around each of its two adjacent faces in a counterclockwise direction. Each edge also keeps information about the incident faces and vertices. It is then possible to follow the list of edges around any face or vertex. The edge data structure thus is part of four of these circular lists, one list for each of the two incident vertices, and one list for each of the two faces. Simple polygonal faces and vertices need have no explicit data structure of their own, and instead simply have a representative incident edge. If the face is allowed to have holes, however, it becomes important to group together the set of representative edges of all holes which are incident to a face, thus requiring a face data structure, and additional pointers from the edges.

A more detailed overview of these two obstacle representations is given in a textbook of O'Rourke [130].

The choice of data structure to be used with the algorithm described in this paper needs to be appropriate for the set of obstacles which are allowed, and additionally needs to be an acceptable input to the chosen partitioning algorithm. Either of the above mentioned models could be made to satisfy these requirements. Because the first choice makes it easier to represent faces with holes, it can be assumed that the input obstacles are represented using the winged-edge data structure, with modification described above to allow for faces with holes.

4.1.2 Allowable Paths

One slightly obscure but nevertheless important consideration is deciding what kinds of paths are allowed in this problem. If two three-dimensional obstacles meet along an edge, then should a path be able to squeeze between them? If the lower face of one obstacle is coplanar with the upper face of another obstacle, but the two faces do not actually meet, can a path travel along both faces without any bends in between?

In the interest of keeping the problem well defined it will be stated, as a matter of definition, that a path can approach an obstacle face in the limit but cannot actually travel at the boundary of an obstacle. This will disallow the squeezing between obstacles described above.

Some applications may permit a path to travel at the boundary of an obstacle. It is straightforward to translate this kind of problem into an equivalent problem which uses the above definition of allowable paths.

4.1.3 Partitioning

The problem of efficiently dividing a polygon into a set of disjoint pieces such that the union of all pieces is equivalent to original polygon is known as the *Partition Problem*.

The partition problem has a long and extensive history spanning well over a century. Indeed its formulation can be traced to the very roots of the field of geometry itself. In his 1902 book that would become the cornerstone of modern geometry, Hilbert proves that, regardless of how a polygon is decomposed, the sum of the areas of its triangular pieces will be equivalent. He further defines this to be the area of the polygon [70]. Recently entire chapters of books have been devoted to the subject of partitioning [61, 32, 131, 85], and it has been a primary focus in the 1980 Ph.D. thesis of Chazelle [30] and in the 1983 Ph.D. thesis of Keil [83].

Many Partitioning Problems can be described by the desired shape of partitioned pieces, and by whether the endpoints of the line segments separating the polygon must lie at the vertices of the polygon. Line segments which satisfy this second requirement are called *diagonals*.

Certain problem formulations may require the line segments which comprise the partition to exclusively be diagonals, or the segments may just be restricted to having endpoints

on the boundary of the polygon. A formulation might also have no restriction at all, thereby permitting *Steiner points*. Steiner points are formed at any interior point of the polygon where two line segments in the solution to the problem intersect.

Triangulation

The special case of the Partitioning Problem where each piece is required to be a triangle is known as the *Triangulation Problem*. Triangulation has a history which is as old and rich as the Partition problem, and it is also discussed in Hilbert's landmark work from 1902 [70].

One of the first triangulation algorithms came in 1911, when Lennes discussed how to triangulate a simple polygon along diagonals in $O(n^2)$ time, thereby proving that such a triangulation must always exist.

Lennes extended his discussion into higher dimensions showing that his result does not generalize into three dimensions. He gave an example polyhedron which cannot be decomposed into tetrahedrons if the tetrahedron vertices must be located at the vertices of the polyhedron. However, he showed that if the tetrahedron vertices are not restricted to polyhedron vertices, then a decomposition is always possible [99].

A series of improvements to the running time of two dimensional triangulation algorithms have been made throughout the years, culminating with an optimal, linear triangulation algorithm given by Chazelle in 1990 [31].

Note that triangulations in two dimensions are typically done along diagonals, and all such triangulations consist of $n - 2$ triangles.

Convex Partitioning

One of the most useful partitions, aside from triangulation, is that which decomposes a polygon into a set of *convex* polygons. A polygon is considered convex if the line segment between any two points is entirely contained within the polygon.

Typically there are two objectives in convex partitioning, first to minimize the running time, and second to minimize the number of convex pieces into which the polygon is divided. Unlike with triangulation, the number of pieces in a convex partition is not constant for a given polygon.

In 1983, Hertel and Mehlhorn described a very straightforward method to find a convex partition of a polygon along diagonals in $O(n)$ time, plus the time to perform a triangulation. This involved starting with a triangulation, and then removing unnecessary line segments from the triangulation one at a time. Chazelle's triangulation algorithm would eventually give this an overall linear running time. This method finds a partition having at most four times the minimum number of pieces [69].

That same year, Greene would describe a method to find a partition along diagonals having the minimum number of pieces. However, this would come at the expensive cost of a $O(n^4)$ running time [61]. In 1985, Keil improved on Greene's running time, showing how to find a convex partition along diagonals having a minimum number of pieces in $O(n^3 \log n)$ time [84].

In another result, Chazelle described how to find a convex partition having the minimum number of pieces in $O(n^3)$ time in his 1980 Ph.D. thesis. The problem considered by Chazelle differs from that in the above results in that it does not require the line segments of the partition to be along diagonals. Indeed these line segments need not touch the boundary of the polygon at all, and may travel between Steiner points [30, 32].

Note that the above results all assume that the input is a simple polygon. If the polygon has holes, then finding a convex partition with the minimum number of pieces is NP-Hard. This true if Steiner points are allowed, as shown by Lingas in 1982 [102], and by Lingas, Pinter, Rivest, and Sharir [105], or if Steiner points are disallowed, as shown by Keil in his 1983 Ph.D. thesis [83].

Rectilinear Partitioning

Fortunately, the convex partitioning problem becomes easier when the polygons under consideration are rectilinear. In this case the only convex polygon is a rectangle, and so any convex partition will be into rectangular pieces. Note, that in this kind of partition it is generally assumed that the dividing segments may have endpoints at non-vertex locations.

Many authors have discussed how to efficiently partition rectilinear polygons with n corners into rectangles. In 1979, Lipski, Lodi, Luccio, Mugnai, and Pagli showed how to partition a rectilinear polygon into a minimum number of rectangles in $O(n^3)$ time [109]. Lipski later improved on this result for rectilinear polygons without holes in 1983, giving an algorithm solving that problem in $O(n^{\frac{3}{2}} \log^2 n)$ time [107, 108].

In 1982, Ohtsuki showed how to partition a rectilinear polygon into at most $n - 2$ rectangles in $O(n \log n)$ time, by placing a horizontal dividing line at every non-convex vertex [128]. See Figure 4.1 for an example of such a partition. Although this may not be the smallest number of rectangles, this number is often sufficiently small, and the improvement in running time is substantial. Additionally, the horizontal dividing segments are a useful feature in many situations [147, 160, 54], including in the proof of Lemma 4.4.8 of this thesis.

A rectilinear polygon with no holes can be partitioned more efficiently. In 1989, Liou, Tan, and Lee showed how to partition a simple rectilinear polygon into a minimum number of rectangles in nearly linear $O(n \log \log n)$ time [106].

Still another objective that has been studied in rectilinear partitioning is that of minimizing the total length of all dividing segments [101, 105].

Rectilinear partitioning is a special concern of this chapter since in the preprocessing stage of the Minimum Link Path algorithm it is necessary to decompose the rectilinear obstacle faces of the problem input into rectangles.

The algorithm described in this chapter assumes that the Ohtsuki algorithm is used for rectilinear partitioning. The main reason for this choice is that the running time improvement is quite substantial, and the number of rectangular pieces does not need to be less than $n - 2$.

4.1.4 Binary Space Partitioning

A *Binary Space Partition (BSP)* is the subdivision of a d-space which contains a set of $d-1$ dimensional objects into smaller d-spaces, such that each of the smaller d-spaces contains at most a constant number of objects.

One kind of binary space partition, called an *auto-partition*, is made by first splitting the entirety of d-space into two pieces along the $d-1$-dimensional hyperplane containing an object. Next, each of the two halves is split again along the $d-1$-dimensional plane containing an object in each of the two halves. This operation continues to recurse until there is at most constant number of objects in every piece.

A 1992 result of Paterson and Yao describes how to find a BSP decomposition of n orthogonal rectangles in three dimensional space in $O(n^{3/2})$ time and space, resulting in a partition which contains $O(n^{3/2})$ fragments of the original rectangles [138]. Their proof has since been simplified by Dumitrescu, Mitchell, and Sharir [44].

A binary space partition is frequently represented as a binary tree. Each node of the tree represents a subspace of the whole, and the two children of a node are the two smaller subspaces into which the larger subspace is divided.

A binary space partition typically results in fragmentation of the objects in the d-space. The size of the BSP is generally considered to be the total number of fragments in the decomposition, although the number of nodes or leaves in the BSP tree has also been used, since all these values are typically within a constant factor of each other.

4.2 Previous $O(n^3)$ Minimum Link Path Algorithms

In this section some existing algorithms solving the Minimum Link Path Problem are discussed. All of these have a worst case running time of $O(n^3)$ when extended into three dimensions.

4.2.1 A Breadth First Search-Based Algorithm

Lee pioneered breadth-first-search routing strategies through two dimensional VLSI circuits in the early 1960's. His method involved first breaking down the plane into a set of $O(n) \times O(n)$ grid cells, as defined by the extension of every obstacle edge, and then marking the cells as they were searched, so as to avoid searching the same cell twice. Using this framework, many techniques applicable to VLSI were developed [93].

Although Lee was primarily concerned with minimizing the forward distance traveled, there exists a similar breadth first search strategy to solve the minimum link path problem in three dimensions in $O(n^3)$ time. Given a set of rectangular obstacle subfaces, define a grid to be the arrangement of all planes which contain an obstacle face. This grid divides space into $O(n^3)$ cells, each of which lies either entirely within an obstacle or entirely outside of all obstacles.

The algorithm works as follows. Mark every grid cell as to whether it is inside of an obstacle, and determine cell adjacencies using some method, such as array indexing. Then proceed in a manner which is nearly identical to a breadth-first search through those cells which lie outside of the obstacles, but with two important differences.

First, proceeding into another cell in the same direction does not increment the distance to that cell, but proceeding in a new direction increments the distance by one. Second, do not immediately prohibit further entry into a cell after it has been visited. Instead, only prohibit entry into that cell via the prior directions of entry. This is necessary since an

optimal path may travel straight through a previously visited cell.

Since there are only two possible bend distances along the frontier of this search, it is not necessary to implement a full priority queue. A bucketed approach such as that described by Dial's Implementation of Dijkstra's algorithm would yield constant insertion and removal times [3].

The breadth-first-search minimum link path algorithm is defined in Algorithm 7. Note that the priority queue holds $\{cell, dir\}$ pairs, and is sorted by the bend distance of that direction in the cell.

Algorithm 7 FIND-BFS-MINLINKPATH$(S, obsFaces, s, t)$

Given: A three-dimensional space S, a list of obstacle faces $obsFaces$ in the space, a starting point s, and a terminating point t
Return: The link-distance between s and t

```
 1: Q ← new PriorityQueue
 2: G ← FIND-GRID(obsFaces, s, t)
 3: FIND-NEIGHBORS(G, obsFaces)
 4: for all ( direction dir ) do
 5:    INSERTPRIORITYQ(Q, {s, dir})
 6: while ( not EMPTYPRIORITYQ(Q) ) do
 7:    {cell, dir} ← REMOVEMINPRIORITYQ(Q)
 8:    newcell ← cell.neighbors.dir
 9:    if ( newcell.isObstacle = FALSE ) then
10:       for all ( direction newdir ) do
11:          if ( newdir ≠ REVERSE-DIR(dir) ) then
12:             if ( newdir = dir ) then
13:                newdist ← cell.benddist.dir
14:             else
15:                newdist ← cell.benddist.dir + 1
16:             if ( newcell.benddist.newdir > newdist ) then
17:                newcell.benddist.newdir ← newdist
18:                INSERTPRIORITYQ(Q, {newcell, newdir})
19: return min(t.benddist.dir) over all directions dir
```

The FIND-BFS-MINLINKPATH algorithm uses the following functions:

- The FIND-GRID function here divides space into $O(n^3)$ cells, along all planes con-

taining an obstacle boundary, and sets the cell $benddist$ variables to ∞, with the exception of s, whose $benddist$ variables are set to 0.

- The FIND-NEIGHBORS function assigns the neighbor relationships between cells, and also marks cells which are along the inside boundary of an obstacle face. Note, it is sufficient to only mark these cells, since this will effectively block all paths into the obstacle.

- The REVERSE-DIR function returns the parallel but opposite direction.

4.2.2 The Mikami-Tabuchi Algorithm

In 1968, Mikami and Tabuchi attempted to improve on the running time of the breadth first search method by reducing the number priority queue operations. They did this with a small modification of breadth first search, which searches through the grid by lines of cells instead of by cells. It is not necessary to insert every cell into the priority queue, if an entire line of cells can be processed between priority queue operations [117].

Although this method drastically reduces the number of priority queue operations, the algorithm still takes $O(n^2)$ time in two dimensions and takes $O(n^3)$ time in three dimensions, since each cell in the grid decomposition must be iterated over. This method can be extended to d dimensions, running in $O(dn^d)$ time.

The method is described in Algorithm 8. The primary difference between this algorithm and the breadth first search algorithm is the addition of a while loop to iterate over all cells in the same line.

Algorithm 8 FIND-MIKAMI-TABUCHI-MINLINKPATH$(S, obsFaces, s, t)$

Given: A three-dimensional space S, a list of obstacle faces $obsFaces$ in the space, a starting point s, and a terminating point t

Return: The link-distance between s and t

```
 1: Q ← new PriorityQueue
 2: G ← FIND-GRID(obsFaces, s, t)
 3: FIND-NEIGHBORS(G, obsFaces)
 4: INSERTPRIORITYQ(Q, {s, 0, NULL})
 5: while ( not EMPTYPRIORITYQ(Q) ) do
 6:    {line, benddist, dir} ← REMOVEMINPRIORITYQ(Q)
 7:    for all ( cell ∈ line ) do
 8:       for all ( direction newdir ) do
 9:          if ( (newdir ≠ dir) and (newdir ≠ REVERSE-DIR(dir)) ) then
10:             newcell ← cell.neighbor.newdir
11:             newline ← NULL
12:             // Iterate over every point in the line
13:             while ( (newcell.isObstacle = FALSE) and
                (newcell.benddist.newdir > benddist) ) do
14:                newline ← newline ∪ newcell
15:                newcell.benddist.newdir ← benddist
16:                // Move to the next position on the line
17:                newcell ← newcell.neighbor.newdir
18:             if ( newline ≠ NULL ) then
19:                INSERTPRIORITYQ(Q, {newline, benddist + 1, newdir})
20: return min(t.benddist.dir) over all directions dir
```

4.2.3 The Sato-Sakanaka-Ohtsuki Tile-Based Algorithm

In 1986, Sato, Sakanaka, and Ohtsuki improved on the running time of existing minimum link path algorithms by eliminating the use of a grid decomposition. Their method instead begins with a rectangular partition of the plane, and searches through the set of rectangles formed by this partition [146].

Using a 1982 technique of Ohtsuki, the plane can be partitioned into rectangles in $O(n \log n)$ time by placing a horizontal line at each convex obstacle vertex [128]. This results in at most $n - 2$ rectangles, called *tiles*, as proven in Lemma 4.4.8.

The priority queue in the Sato-Sakanaka-Ohtsuki algorithm contains the horizontal boundaries between tiles, as well as horizontal lines through the starting and finishing points. Each such boundary stores information about the kinds of optimal paths that can cross it. For a given boundary there can only be two possible optimal link-distances, but this could alternate $O(n)$ times across the interval. See Figure 4.1.

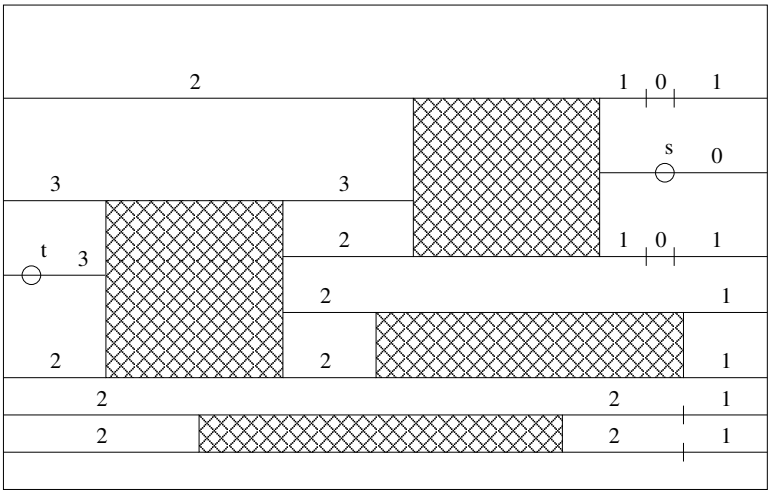


Figure 4.1: Bend distances to tile boundaries in the Sato-Sakanaka-Ohtsuki Algorithm

The set of boundary objects form a graph, where the neighbors of a boundary are all the other boundaries adjacent to the same tile. Thus each boundary has a number of same side neighbors, and a number of opposite side neighbors within a tile (recall that there are

no vertical boundaries). The set of neighbors can be determined in linear time by a 1984 technique of Ousterhout [132].

The algorithm begins with the set of all possible optimal paths lying along the line through the starting point. This includes a set of paths which begins by traveling horizontally, and a second set which begins by traveling vertically. The priority queue is initialized with these paths.

The algorithm proceeds to remove items from the head of the priority queue. The optimal paths to neighboring boundaries adjacent to the same tile are determined by looking at the direction of the incoming paths, and which side the neighbor is on. Horizontal paths can reach same side neighbors without additional bends, and vertical paths can reach opposite side neighbors within the same range without additional bends. All other paths require one additional bend. The resulting paths are inserted back into the priority queue.

One issue to be resolved in this method is determining how sets of paths entering through multiple boundaries of a tile can be merged onto a single exit boundary. The authors suggest that 2-3 trees can be used without going into details about this operation. In 1992, Yang, Lee, and Wong [160] explained in detail that these operations can be performed efficiently, adopting a variation on the segment tree structure which had been described by Imai and Asano [75]. This requires $O(n \log n)$ space, although if 2-3 trees were used then $O(n)$ space would be required. In either case, the Sato-Sakanaka-Ohtsuki algorithm runs in $O(n \log n)$ time.

In 2001, Fitch, Butler, and Rus extended the Sato-Sakanaka-Ohtsuki method into three dimensions, giving an algorithm which iterates over $O(n)$ instances of the Sato-Sakanaka-Ohtsuki algorithm simultaneously. Their method solves the three dimensional problem in $O(n^2 \log n)$ time in many cases, but still has an $O(n^3)$ worst case running time [54].

4.2.4 The Suzuki-Ohtsuki-Sato Line-Based Algorithm

In 1987, Suzuki, Ohtsuki, and Sato developed another method to solve the Minimum Link Path Problem. This was based on searching through a set of line segments, like with the Mikami-Tabuchi Algorithm. Their idea was to place one line segment running parallel to and along each obstacle edge, located just outside the obstacle. The line segment extends in both directions until it reaches an obstacle. See Figure 4.2.

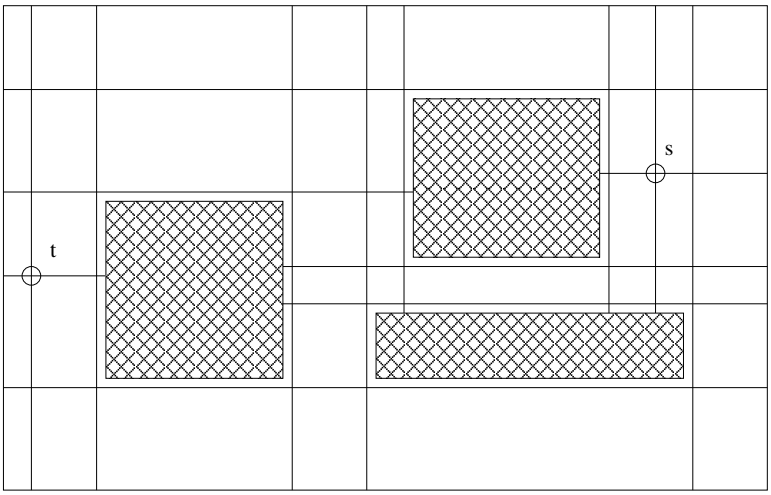


Figure 4.2: The set of lines to search along in the Suzuki-Ohtsuki-Sato Algorithm

The idea here is that there must be an optimal path which travels exclusively along these line segments, so an optimal path can be found by a breadth first search through this set of line segments.

A fundamental problem this algorithm must then solve is to return the set of horizontal line segments which intersect a query vertical line segment. The authors mention that this problem can be solved using a priority search tree, resulting in an overall algorithm which takes $O(n \log^2 n)$ time and $O(n)$ space. Alternately, the problem can be solved with a segment tree, yielding an algorithm taking $O(n \log n)$ time and $O(n \log n)$ space [154].

Extending this method into higher dimensions has not been done previously, but a possible method is mentioned in Section 7.2.5, and is considered for future work.

4.3 Overview of a new $O(\beta n \log n)$ Binary Space Partition-Based Algorithm

Here we introduce the first algorithm which improves on the worst case running time of the three dimensional minimum link path problem. The algorithm described here follows a similar approach to the $O(n^3)$ algorithms described previously, but saves time by using a binary space partition decomposition of space [138, 44], instead of a grid decomposition or a tiling.

The binary space partition is represented with a tree, where the leaves of the tree all designate subspaces which are either entirely within an obstacle or entirely outside obstacles. We refer to the subspace represented by such a leaf as a *block*. We call the blocks outside of obstacles *empty blocks* and the blocks which are within an obstacle *obstacle blocks*. The problem then reduces to searching for an optimal path through the set of empty blocks.

The algorithm finds all points reachable by paths with zero bends, then by paths with one bend, then with two bends, and so on, until the destination point is found.

These paths are found using a series of sweep plane operations. Begin with six sweeps, each of which follows the zero-bend path from the starting point in a particular axis-parallel direction. Then perform six more sweeps, each following the one-bend paths whose last segment is in a particular direction, and so on.

The sweep operation for a bend distance b and direction $\mathcal{D}$ keeps track of all points on the sweep plane that are reachable in exactly b bends, with the last segment traveling in direction $\mathcal{D}$. As the plane sweeps across the various blocks, it is updated by adding or removing regions of reachable points. When the plane encounters an obstacle we must remove any points that lie in the region corresponding to the obstacle face. When the plane leaves an empty block which has been encountered before, we must add any previously saved outgoing paths of bend distance b to the sweep plane.

Whenever the sweep plane encounters an empty block which does not already have incoming paths of bend distance less than $b-3$, we determine the best paths from the entry face to every point on all six exit faces. We store these paths as events for future sweep operations.

The remainder of this chapter will be concerned with this algorithm.

4.4 Preprocessing

Preprocessing for the algorithm takes place in several stages. First, all rectilinear faces of all obstacles are partitioned into rectangles, which we call *subfaces*. Second, this set of rectangles is used to define a *binary space partition* of the space containing the obstacles. The binary space partition may fragment the subfaces into smaller pieces which we call *subface fragments*, and partitions space into smaller rectangular pieces which we call *blocks*. Third, all pairs of neighbors in the binary space partition need to be established.

Each of these successive refinements is defined here.

Definition 4.4.1 (face) *Define a* face *to be the planar boundary of an obstacle which lies within one contiguous region, and is represented by a the face data structure of the obstacle model. Within this problem all faces are rectilinear.*

Definition 4.4.2 (subface) *Define a* subface *to be any rectangular piece of the convex (rectangular) partition of a rectilinear obstacle face.*

Definition 4.4.3 (subface fragment) *Define a* subface fragment *to be a piece of a subface resulting from fragmentation by a binary space partition.*

The total number of corners in all faces is n, as per the definition of n given in Chapter 1, so the total size of all faces is $O(n)$. The number of subfaces is also $O(n)$, as shown in

Lemma 4.4.8, and the number of subface fragments is β as per the definition of the size of a binary space partition tree given by Dumitrescu, Mitchell, and Sharir [44]. The value of β is bounded by $O(n^{3/2})$ as shown by both Paterson and Yao [138] and Dumitrescu, Mitchell, and Sharir [44].

Data Structure	Total size of all instances
Faces	$O(n)$
Subfaces	$O(n)$
Subface Fragments	$\beta = O(n^{3/2})$

Table 4.1: The sizes of all faces, subfaces, and subface fragments

4.4.1 Partitioning into Subfaces

The first step of preprocessing is to partition all rectilinear obstacle faces into rectangles. This is done using the Ohtsuki method described in Section 4.1.3. The partition of the surface of a three dimensional rectilinear polygon having n corners using Ohtsuki's method results in at most $n - 2$ rectangles in each dimension as shown in Lemma 4.4.8. All dividing line segments of this partition are horizontal (as defined in a consistent way for each dimension). Ohtsuki's partition algorithm takes $O(n \log n)$ time [128].

Several relations between the number of subfaces and n, the number of obstacle corners are established here.

Definition 4.4.4 **(c_i)** *Let c_i be the total number of times a corner of a subface perpendicular to the i-axis is an original corner of the polygon before partitioning.*

Lemma 4.4.5 $\forall\, i \in \{x, y, z\}\ (c_i = n)$

Proof. Every one of the original n corners is the vertex of exactly one subface in each dimension, so $c_x = c_y = c_z = n$. □

Definition 4.4.6 **(f_i)** *Let f_i be the number of subfaces perpendicular to the i-axis.*

Lemma 4.4.7 $\forall\, i \in \{x, y, z\}\ (f_i \geq n/4)$

Proof. As before, we have $\forall\, i \in \{x, y, z\}\ (c_i = n)$. Also any subface has at most four of the original corners at its vertices, so $\forall\, i \in \{x, y, z\}\ (c_i \leq 4f_i)$ and thus $\forall\, i \in \{x, y, z\}$ $(f_i \geq n/4)$. □

Lemma 4.4.8 $\forall\, i \in \{x, y, z\}\ (f_i \leq n - 2)$.

Proof. Every subface of a given dimension can be associated with a unique corner of the original polygon as follows. For any subface f, begin at the corner of f which has the maximal coordinate in both directions (the top right corner).

This corner has a vertical line segment adjacent to it on the negative side (extending below). Recall that Ohtsuki's partitioning algorithm uses only horizontal dividing segments, so this vertical line segment must be an edge of the original polygon.

This corner also has a horizontal line segment adjacent to it on the negative side (extending to the left). This might be an edge of the original polygon or it might be a dividing line segment, added by Ohtsuki's partition. If it is an original edge, then the two segments meet at an original corner, and this corner is then associated with the subface f.

If the horizontal edge is a dividing edge, then one of the endpoints must be a corner of the original polygon, since Ohtsuki's algorithm only adds dividing segments with at least one endpoint at a corner. Associate this original corner with f. If both endpoints are corners, associate the right endpoint with f.

Finally, it remains to show that each vertex is uniquely associated with a face. That is, no vertex can be associated with more than one subface.

Assume, by way of contradiction, that a vertex is associated with two subfaces. Recall that each face selected a vertex along its top edge. So then the only way a vertex could

be associated with two subfaces is if the same vertex is selected as the top left corner of one face and as the top right corner of another face. This then implies that the two faces are separated by some kind of vertical line. This cannot be a dividing segment, since there are only horizontal dividing line segments. Additionally, this cannot be the boundary of the polygon, since both sides of the line are subfaces and thus are interior to the polygon. This is not allowed by the obstacle representation. Therefore there can be no vertical line between the two faces, and there is a contradiction.

Thus, every subface in a particular dimension can be uniquely associated with at least one of the original n corners and $\forall\, i \in \{x, y, z\}\ (f_i \leq n)$.

Additionally, there are at least two corners along the bottom edge which are not associated with any subface, and so $\forall\, i \in \{x, y, z\}\ (f_i \leq n - 2)$. □

Corollary 4.4.9 *The Rectilinear Partition of all faces described above results in at most* $3(n-2)$ *subfaces.*

Corollary 4.4.10 $\forall\, i \in \{x, y, z\}\ (n/4 \leq f_i \leq n)$

Corollary 4.4.11 $\forall\, i, j \in \{x, y, z\}\ (f_i \leq 4f_j)$

4.4.2 Binary Space Partitioning

Given a set of rectangular obstacle subfaces, the second step is to compute a Binary Space Partition of the space containing these subfaces, using the method described by Paterson and Yao [138]. This decomposes space into a set of regions, which we call *blocks*, and divides the subfaces into subface fragments with the following properties.

Property 4.4.12 *Each block corresponds to a leaf in the tree representation of a BSP.*

Property 4.4.13 *No block contains any part of a subface.*

Property 4.4.14 *No subface fragment crosses the boundary of any partition in the entire binary space partition.*

Note, the starting and ending points are considered zero-dimensional obstacles, and so each gets its own leaf.

Lemma 4.4.15 *A binary space partition (BSP) of the space containing a set of rectilinear obstacles divides space into a set of blocks, such that each block lies either entirely within an obstacle, or is entirely outside of any obstacles.*

Proof. This follows from Property 4.4.13. □

Corollary 4.4.16 *The question of whether a block is inside an obstacle or outside of all obstacles can be entirely determined by looking at which side it lies on the splitting face of its parent in the tree representation of the BSP.*

It is therefore possible to label the blocks, in constant time per block, as empty blocks and as obstacle blocks, where an obstacle block lies entirely within an obstacle, and an empty block lies entirely outside of any obstacle.

Lemma 4.4.17 *Any space containing n orthogonal rectangles can be subdivided with a binary space partition resulting in at most $O(n^{3/2})$ rectangle fragments.*

Proof. Proofs have been given by Paterson and Yao [138] and by Dumitrescu, Mitchell, and Sharir [44]. □

Lemma 4.4.18 *The number of blocks resulting from a BSP of space, using the Paterson and Yao method, is at most one more than the number of subface fragments.*

Proof. In the tree representation of the binary space partition, each leaf node represents one of the blocks, and each internal node represents a partitioning rectangle.

The Paterson and Yao partitioning method only partitions along subface fragments, so there is at least one fragment per internal node.

Since this is a binary tree the number of internal nodes is one less than the number of leaves. □

Theorem 4.4.19 *Space can be subdivided into $O(\beta)$ or $O(n^{3/2})$ blocks, including both obstacle blocks and empty blocks, by a Binary Space Partition.*

Proof. This follows from lemmas 4.4.18 and 4.4.17 □

4.4.3 Neighbors in a Binary Space Partition

In the implementation of the algorithm described later, it will be necessary to know which blocks share a face with each other in space. We will call such pairs neighbors. The running time of preprocessing in the algorithm will be dominated by the amount of time expended building a graph of all neighbor relationships. The time spent in sweep plane operations also depends in part on the size of the neighbor graph.

Consider the set of all edges surrounding all obstacle subfaces. The following definition is found in Paterson and Yao [138]:

Definition 4.4.20 **(p_i)** *Let p_i to be the number of edges running parallel to the i-axis, for $i \in \{x, y, z\}$. Count four edges for each subface, even though edges may be shared.*

Without loss of generality, assume the following property:

Property 4.4.21 $p_x \geq p_y \geq p_z$.

Recalling the definition of f_i from Definition 4.4.6, it follows that:

Property 4.4.22 $p_x = 2f_y + 2f_z$

Property 4.4.23 $p_y = 2f_x + 2f_z$

Property 4.4.24 $p_z = 2f_y + 2f_x$

This then yields the following theorem.

Theorem 4.4.25 $p_x \leq 4p_z$

Proof. From Corollary 4.4.11 we have $2f_y + 2f_z \leq 10f_y$ and $\frac{5}{2}f_y \leq 2f_x + 2f_y$. Therefore $p_x \leq 10f_y = 4(\frac{5}{2}f_y) \leq 4p_z$. □

Definition 4.4.26 (i-cut) *Define an i-cut to be any of the rectangles which divided a region of space during the course of a BSP decomposition, and which is perpendicular to the i-axis for $i \in \{x, y, z\}$.*

Any edge in the tree representation of a BSP corresponds to an i-cut. A path from the root of the tree to a leaf will thereby travel through a series of x, y, and z-cuts. The following results come from Paterson and Yao [138].

Lemma 4.4.27 *The number of x-cuts along any path from the root to a leaf is at most* $\max\{0, \frac{1}{2}(\lg(11p_yp_z/p_x))\}$.

Proof. This is proven by Paterson and Yao [138]. □

Lemma 4.4.28 *The number of y-cuts along any path from the root to a leaf is at most* $\max\{\lg(p_x/p_y), \frac{1}{2}(\lg(11p_xp_z/p_y))\}$.

Proof. This is proven by Paterson and Yao [138]. □

Lemma 4.4.29 *The number of z-cuts along any path from the root to a leaf is at most* $\max\{\lg(p_x/p_z), \frac{1}{2}(\lg(11p_xp_y/p_z))\}$.

Proof. This is proven by Paterson and Yao [138]. □

This then gives us the following:

Lemma 4.4.30 *The number of i-cuts and j-cuts, where $i, j \in \{x, y, z\}$, along any path from the root to a leaf of a BSP-Tree decomposition of the space containing our obstacles is at most* $\lg(11p_x)$.

Proof. By Theorem 4.4.25 and Property 4.4.21 we have $4p_z \geq p_x \geq p_y \geq p_z$, which yields:

$$0 < \frac{1}{2}(\lg(11p_yp_z/p_x))$$

$$\lg(p_x/p_y) < \frac{1}{2}(\lg(11p_xp_z/p_y))$$

$$\lg(p_x/p_z) < \frac{1}{2}(\lg(11p_xp_y/p_z))$$

Thus, in the maximum calculations from Lemmas 4.4.27 through 4.4.29, the larger value is the second of the two choices. Also, by Property 4.4.21 we get:

$$\frac{1}{2}(\lg(11p_yp_z/p_x)) \leq \frac{1}{2}(\lg(11p_xp_z/p_y)) \leq \frac{1}{2}(\lg(11p_xp_y/p_z))$$

Therefore, the quantity in this lemma is at most

$$\frac{1}{2}(\lg(11p_xp_z/p_y)) + \frac{1}{2}(\lg(11p_xp_y/p_z))$$

$$= \frac{1}{2}(\lg 11 + \lg p_x + \lg p_z - \lg p_y) + \frac{1}{2}(\lg 11 + \lg p_x + \lg p_y - \lg p_z)$$

$$= \lg 11 + \lg p_x$$

$$= \lg (11 p_x)$$

□

Lemma 4.4.31 *A plane can pass through the interior of at most $O(n)$ blocks of our BSP decomposition of space.*

Proof. Consider a plane that passes through the BSP decomposition perpendicular to the i-axis. The set of blocks which it intersects can be found by following all branches of the BSP-Tree except branches of i-cuts which face away from the plane in question. This pruned tree has the same number of leaves as a tree which can have cuts in only two of the three dimensions. By Lemma 4.4.30, the number of cuts along any path from the root to a leaf of such a tree is at most $\lg p_x$ plus a constant. Since $p_x = O(n)$ this tree has $O(n)$ leaves. □

Theorem 4.4.32 *There are at most $O(\beta n)$ neighbor relationships between pairs of blocks, and between empty blocks and obstacle subfaces.*

Proof. By Lemma 4.4.31 each block can have at most $O(n)$ neighboring blocks across a given face. Furthermore, there are $O(n)$ obstacle subfaces by Theorem 4.4.8, so each block can have at most $O(n)$ neighboring subfaces. The total number of blocks is $O(\beta)$ as given in Theorem 4.4.19 which limits the total number of neighbor relationships to $O(\beta n)$. □

We will want to find all neighbors of an empty block B across a particular block face, F. We can find all blocks incident to the plane containing F by traversing the pruned tree

described in the proof of Lemma 4.4.31, testing each to see if it is a neighbor of F. We can find all obstacle subfaces incident to this plane by simply testing each one in turn.

Theorem 4.4.33 *The total time to determine all neighbors is* $O(\beta n)$.

4.5 Paths through Blocks

Define the bend distance to a point in space and to a block as follows.

Definition 4.5.1 (Bend Distance (to a point)) *Define the* bend distance *to a point* p *in space to be the number of bends in a minimum link path between the starting point* s *and* p.

Definition 4.5.2 (Bend Distance (to a block)) *Define the* bend distance *to a block* B *to be the smallest number of bends in a minimum link path between the starting point* s *and any point* $p \in B$.

Within a single empty block, there can be several variations in the optimal bend distance to different places within the block. This section examines the kinds of optimal paths which can travel through an empty block.

An examination of a block must answer the following question: "Given the set of points on a single face of a block through which optimal paths can enter with a particular bend distance, what configuration of exit points could be generated by the continuations of those optimal paths, and what are the optimal bend distances to those points?"

4.5.1 Classification of Paths through a Block

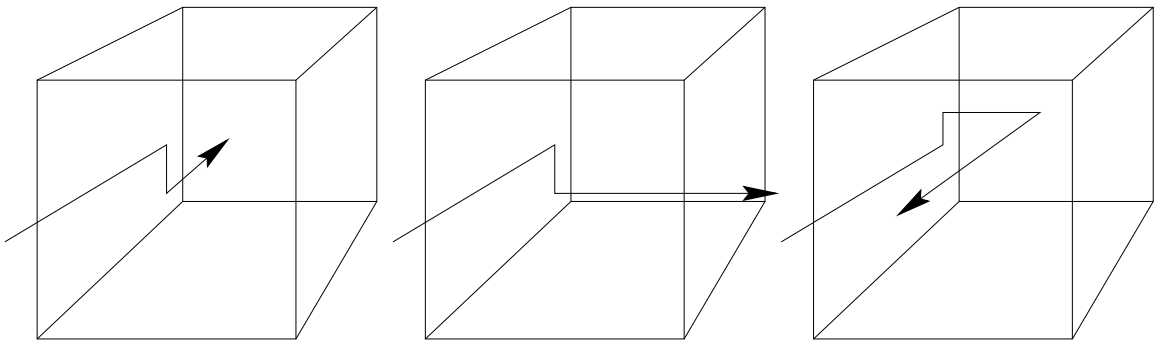
Define three kinds of paths which could travel through a block, based on the choice of their exit face:

Definition 4.5.3 (Through Paths) *Paths exiting the block through the face opposite to the one through which they entered.*

Definition 4.5.4 (Right Angle Paths) *Paths exiting the block through one of the four faces which are not parallel to the entry face (We use "Right Angle" here to indicate the angle between the entry and exit faces).*

Definition 4.5.5 (U-Turn Paths) *Paths exiting the block through the same face through which they entered.*

Figure 4.3 gives some examples of these paths.



(a) (b) (c)

Figure 4.3: Three kinds of paths: (a) a Through Path, (b) a Right Angle Path, and (c) a U-Turn Path

We also classify three configurations of exit points. Table 4.2 describes the configurations which could be reachable on an exit face with a certain number of bends from a single entry point.

Table 4.3 describes all relationships between the exit face of a path, the number of bends the path takes within a block, and the configuration of exit points. Any paths with additional bends beyond those listed in this table will be suboptimal.

Class	Point Configuration on Exit Face
`Class A`	Entire Exit Face
`Class B`	Line Segment Spanning Exit Face
`Class C`	Single Point on Exit Face

Table 4.2: Three classes of exit point configurations from a single point of entry

# of bends	0	1	2	3
Through Paths	C	-	B	A
Right Angle Paths	-	B	A	-
U-Turn Paths	-	-	B	A

Table 4.3: The relationships between path type, exit point configuration, and number of bends in an optimal path through a block

4.5.2 Generating Exit Paths

Given an entry face with a set of entry points, we would like to generate the appropriate sets of exit points resulting from the paths described above. All optimal paths through a block generate one of the configurations A, B, or C, so it is sufficient to only consider these classes.

- `Class A` point configurations cover an entire exit face. Thus, from any entry point or points, the complete rectangular outgoing face of the block is the data generated.

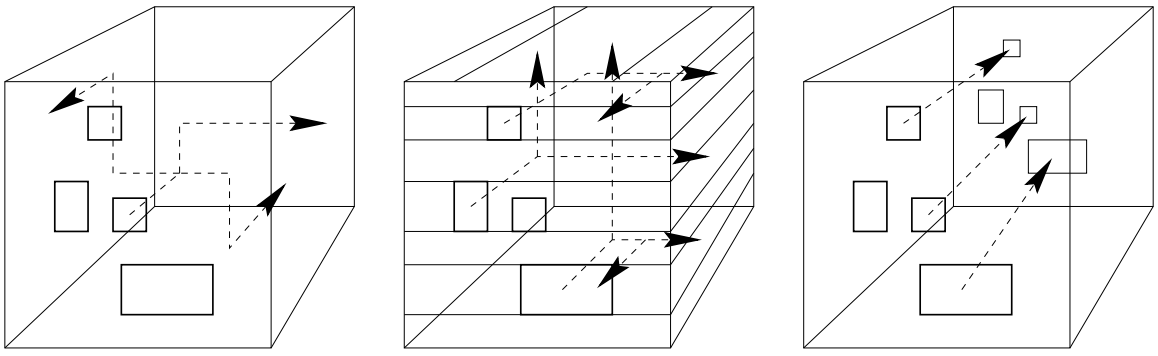
- `Class B` exit points resulting from a single point of entry lie in one of the line segments formed at the intersection of a plane containing the entry point and an exit face. The paths which lead to these points must remain within one of the two planes perpendicular to the face which contains the entry point.

  The `Class B` exit points associated with all points entering through an entry face and exiting through a specific exit face is the collection of all parallel line segments which are defined by the intersection of the plane of a point of entry and the exit face. Another way to visualize this is to project the entry data onto one of its axes,

and then sweep the resulting set of intervals across the outgoing face, creating a set of stripes which span the exit face.

All six faces have `Class B` points exiting them. The entry face, and the face opposite it, each have two sets of `Class B` points exiting through them, one set striped in each direction, each perpendicular to the direction of entry into the block. The remaining faces can only be striped in a single direction, parallel to the direction of entry into the block.

- `Class C` exit points are identical in configuration to the set of incoming points. This data could be complex, so copying it in its entirety at every block would be prohibitively time consuming. We therefore employ a sweep plane, described in Section 4.7, which carries the same data forward from block to block without copying.



(a) (b) (c)

Figure 4.4: Some examples of the three configurations of exit points: (a) `Class A`, (b) `Class B`, and (c) `Class C`.

4.6 Priority Queue

During processing we will maintain a priority queue of events to be handled by the sweep plane. Here we describe how this priority queue is maintained.

The priority queue is initialized with the set of outgoing paths leaving the starting block. The algorithm then proceeds much like a Breadth First Search, with events being pulled off the head of the priority queue and processed, and new events generated which are inserted back into the queue.

The priority queue may contain the following types of events:

- **[**`OutgoingPath`**]** A set of paths leaving the face of a empty block
- **[**`ObstacleFace`**]** An obstacle face
- **[**`EmptyBlock`**]** The entry face into an empty block

Regardless of the type of event, all events share some common data. The following shared data will be used, in order of priority, to sort the priority queue:

1. **[**$benddist$**]** Minimum bend distance to the event
2. **[**dir**]** Direction
3. **[**$coord$**]** Coordinate in the direction (front to back)
4. **[**$eventType$**]** Event type (in the order listed above)

Operations on the sweep plane are triggered by events as they are removed from the priority queue. When an `OutgoingPath` event is removed, a set of paths is inserted into the sweep plane using the INSERTSTRIPEDRECT function. When an `ObstacleFace` event is removed, a rectangle is cleared from the sweep plane using the CLEARRECT function. These operations will be described later in Section 4.7.

The standard operations on a priority queue are used in support of this:

- INSERTPRIORITYQ$(Q, eventType, eventData)$

  Insert an event or set of events into the priority queue.

- REMOVEMINPRIORITYQ(Q)

  Remove the smallest event from the priority queue, returning it.

- EMPTYPRIORITYQ(Q)

  Return `TRUE` if the priority queue is empty.

The Event data structure contains the following information:

```
Event =
   EventType: eventType
   Int: benddist
   Direction: dir
   Num: coord //Coordinate in direction dir
   Num: top, bottom, left, right //Block or face bounds
   Block: block //The block associated with this event
   Face: face //For face events, the associated face
   SegTree: data //For outgoing path events, the path data
   Axis: stripeDir //The direction of the striping of data
```

The same empty block or obstacle face may be added to the priority queue multiple times within the same sweep from different neighbors. So if two such identical events are added to the priority queue, the second is discarded.

4.7 Sweep Plane

As outlined in Section 4.3, we will maintain a sweep plane during each of a series of sweep operations. The regions stored in the plane contain points which are reachable by paths of a

particular bend distance, and whose last segment is in the direction of the sweep. We store this data as a two-dimensional segment tree.

4.7.1 Two-Dimensional Segment Trees

A data structure is needed to describe the sweep plane, as well as the sets of paths which can enter and exit an empty block. A two-dimensional segment tree is employed for these. This data structure is designed to allow the insertion and deletion of rectangles, and to support efficient queries. The advantage of using this data structure is that a set of rectangles can be stored efficiently, even if the rectangles are criss-crossing and overlapping.

We define a variation on the traditional segment trees. This kind of two-dimensional segment tree does not distinguish between rectangles after they have been inserted. Rather the union of all inserted regions is kept.

This modification allows for some additional efficient operations, including clear, projection and sweep or stripe functions.

The following operations are defined on this new kind of segment tree:

- INSERTRECT inserts a rectangle into the tree.
- INSERTSTRIPEDRECT inserts a set of $O(n)$ stripes which are contained within, and extend the length or width of a given rectangle. The stripes vary in the other dimension according to the set of intervals defined in a given one-dimensional segment tree.
- QUERYRECT queries a rectangle in the segment tree, returning `TRUE` if it overlaps with stored data.
- CLEARRECT clears all data within the specified rectangle (Note this may fragment previously inserted rectangles).

- PROJECTRECT projects all data within the given rectangle onto an axis, returning a one-dimensional segment tree containing the resulting intervals.

Some auxiliary data is maintained in the nodes of the two-dimensional segment tree in order to support efficient projections and queries. The above operations then each run in amortized $O(n \log n)$ time or better. The implementation of these operations and the proof of their running times is described in Chapter 5.

4.7.2 Events

There are three types of events that the sweep may encounter. Here a SWEEPPLANEEVENT function is defined. This processes these events based on the event type as follows:

- `OutgoingPath`: A set of paths leaving a block in the direction and bend distance of the sweep are encountered. Add them to the sweep plane, by calling INSERTRECT or INSERTSTRIPEDRECT, for Class A and B points respectively. Neighboring empty blocks and obstacles are inserted into the queue as `EmptyBlock` events and as `ObstacleFace` events, respectively.
- `ObstacleFace`: An obstacle subface is encountered. All paths which intersect that face must be deleted from the sweep plane, using CLEARRECT.
- `EmptyBlock`: An empty block is encountered. If it was discovered three bends or more previously, treat it like an obstacle, calling CLEARRECT, since all optimal paths through the block have already been generated. Note, this is necessary to prevent paths from leaking through to the other side where there may be obstacles which will go unprocessed. If it had been discovered less than three bends ago, query the sweep plane, via a call to QUERYRECT, to see if any paths enter that block. If so, then outgoing paths are generated, using PROJECTRECT, and inserted into the priority

queue as `OutgoingPath` events. Neighboring empty blocks and obstacle subfaces are inserted into the queue as `EmptyBlock` events and as `ObstacleFace` events, respectively.

The algorithm SWEEPPLANEEVENT processes a single event encountered by the sweep plane. It takes advantage of the following functions.

- CLASSAPATHS, CLASSBPATHS: Return the set of Class A or Class B paths exiting the given block, for the given set of incoming paths, as described in Section 4.5.2. Increment $benddist$ by 1, 2, or 3 bends as appropriate, and assign the $left$, $right$, $bottom$, and top relative to the new direction of approach.

Algorithm 9 SWEEPPLANEEVENT(Q, $Plane$, ev, $benddist$, dir)

Given: The priority queue Q, the sweep plane $Plane$, an event ev, the current bend distance $benddist$, and direction of the sweep dir.

```
1: if ( ev.eventType = EmptyBlock ) then
2:    if ( ev.block.benddist > benddist − 3 ) then
3:       if ( QUERYRECT( Plane, ev.left, ev.right, ev.bottom, ev.top) ) then
4:          INSERTPRIORITYQ(Q, OutgoingPath, CLASSAPATHS(ev.block, Plane))
5:          INSERTPRIORITYQ(Q, OutgoingPath, CLASSBPATHS(ev.block, Plane))
6:          INSERTPRIORITYQ(Q, EmptyBlock, {ev.block.neighbors.dir, benddist,
            dir})
7:          INSERTPRIORITYQ(Q, ObstacleFace, {ev.block.obsneighbors.dir,
            benddist, dir})
8:    else if ( ev.block.benddist ≤ benddist − 3 ) then
9:       CLEARRECT(Plane, ev.left, ev.right, ev.bottom, ev.top)
10: else if ( ev.eventType = ObstacleFace ) then
11:    CLEARRECT(Plane, ev.left, ev.right, ev.bottom, ev.top)
12: else if ( ev.eventType = OutgoingPath ) then
13:    INSERTSTRIPEDRECT(Plane, ev.data, ev.left, ev.right, ev.bottom, ev.top,
       ev.stripeDir)
14:    INSERTPRIORITYQ(Q, EmptyBlock, {ev.block.neighbors.dir, benddist, dir})
15:    INSERTPRIORITYQ(Q, ObstacleFace, {ev.block.obsneighbors.dir,
       benddist, dir})
```

Theorem 4.7.1 *The* SWEEPPLANEEVENT *function takes amortized* $O(n \log n + m \log n)$ *time, where* m *is the number of neighbors of the block associated with the given event.*

Proof. The SWEEPPLANEEVENT function makes a constant number of CLASSAPATHS, CLASSBPATHS, CLEARRECT, and INSERTSTRIPEDRECT calls. The number of INSERTPRIORITYQ calls is a constant plus at most the number of neighbors of the block associated with this event. The running time of these are examined in turn.

- [INSERTPRIORITYQ]: The INSERTPRIORITYQ call takes $O(\log n)$ time by the running time of priority queue operations. It is called a constant number of times plus at most once per neighbor.
- [CLASSAPATHS, CLASSBPATHS]: The CLASSAPATHS function calls QUERYRECT and INSERTRANGE, and the CLASSBPATHS function calls PROJECTRECT. Each of these takes amortized $O(n \log n)$ time as shown in Chapter 5.
- [CLEARRECT, INSERTSTRIPEDRECT]: The CLEARRECT and INSERTSTRIPEDRECT calls take amortized $O(n \log n)$ time as shown in Chapter 5.

Therefore, SWEEPPLANEEVENT takes amortized $O(n \log n + m \log n)$ time. □

4.8 Algorithm

The main result of this chapter appears as Algorithm 10. Here we analyze the time and space requirements of this algorithm. Note the following helper functions are used in the algorithm.

- INITIALIZEPRIORITYQ: Insert into the priority queue `OutgoingPath` events leaving the starting block and having zero bends. Insert any neighboring blocks as

`EmptyBlock` events, and any neighboring obstacles subfaces as `ObstacleFace` events.

- RECT-PARTITION: Run the rectilinear partition algorithm of Ohtsuki.
- FIND-BSP: Run the binary space partition algorithm of Paterson and Yao.
- TREELEAVES: Return the set of all leaves of a tree.
- FIND-NEIGHBORS: Compute the neighbor graph, using the method described in Section 4.4.3, and update the neighbor list in each block.

Algorithm 10 FIND-MINLINKPATH$(S, obsFaces, s, t)$

Given: A three-dimensional space S, a list of obstacle faces $obsFaces$ in the space, a starting point s, and a terminating point t
Return: The link-distance between s and t

```
 1: Q ← new PriorityQueue
 2: sweepPlane ← new SegTree2D
 3: subfaces ← RECT-PARTITION(obsFaces, s, t)
 4: bsptree ← FIND-BSP(subfaces)
 5: blocks ← TREELEAVES(bsptree)
 6: FIND-NEIGHBORS(bsptree, blocks, subfaces)
 7: INITIALIZEPRIORITYQ(Q, s)
 8: benddist ← 0; dir ← 0
 9: while ( not EMPTYPRIORITYQ(Q) ) do
10:    event ← REMOVEMINPRIORITYQ(Q)
11:    if ( event.benddist ≠ benddist or event.dir ≠ dir ) then
12:       CLEARRECT(sweepPlane, sweepPlane.minx, sweepPlane.maxx,
          sweepPlane.miny, sweepPlane.maxy)
13:       benddist ← event.benddist
14:       dir ← event.dir
15:    SWEEPPLANEEVENT(Q, sweepPlane, event, benddist, dir)
16: return t.benddist
```

4.8.1 Runtime Analysis

Here the running time of Algorithm SWEEPPLANEEVENT is examined.

Theorem 4.8.1 *The number of events removed from the priority queue is* $O(\beta)$.

Proof. Since duplicate events are deleted from the priority queue, an event is only removed from the priority queue once per unique combination of event block, event bend distance, event direction, and event type. The number of combinations is proportional to the number of blocks, $O(\beta)$, since the same block is not inserted after a constant number of bends later than its first insertion, and there are a constant number of event types and directions. □

The running time of each call in the algorithm is examined in turn.

- [INITIALIZEPRIORITYQ]: Initializing the priority queue takes $O(m \log n)$ time, where m is the number of neighbors of the starting block. It involves the insertion of a constant number of `OutgoingPath` events, taking $O(\log n)$ by the cost of priority queue operations. Additionally it requires an `EmptyBlock` or `ObstacleFace` event to be inserted for each neighbor of the starting block.

- [RECT-PARTITION]: The RECT-PARTITION function takes $O(n \log n)$ time using the method of Ohtsuki, resulting in at most $n - 2$ subfaces [128].

- [FIND-BSP]: A binary space partition of n subfaces takes $O(n^{3/2})$ time using the method of Paterson and Yao, resulting in $O(n^{3/2})$ blocks [138].

- [TREELEAVES]: Walking through an entire tree takes time proportional to its size. The size of the binary space partition tree being walked is $O(\beta)$=$O(n^{3/2})$, by the outcome of the binary space partition algorithm [138].

- [FIND-NEIGHBORS]: The time to find all neighbors is $O(\beta n)$ by Theorem 4.4.33.

- [SWEEPPLANEEVENT]: The SWEEPPLANEEVENT function takes amortized $O(n \log n + m \log n)$ time per call, where m is the number of neighbors of the

block associated with the given event, by Theorem 4.7.1. It is called once every time an event is removed from the priority queue. The number of events removed from the priority queue is $O(\beta)$, by Theorem 4.8.1. The total number of neighbors relationships is $O(\beta n)$, by Theorem 4.4.32. Therefore, the total time spent in SWEEPPLANEEVENT calls is $O(\beta n \log n)$.

- [CLEARRECT]: The CLEARRECT function takes amortized $O(n \log n)$ time per call, as shown in Chapter 5, and is called at most once every time an event is removed from the priority queue. The number of events removed from the priority queue is $O(\beta)$, by Theorem 4.8.1. Therefore, the total time spent in CLEARRECT calls is $O(\beta n \log n)$.

Theorem 4.8.2 *The algorithm described in this chapter finds the Link-Distance between s and t in $O(\beta n \log n)$ time.*

4.8.2 Space Requirements

Theorem 4.8.3 *The space requirements of this algorithm are $O(\beta n)$ or $O(n^{5/2})$.*

Proof. The size of each data structure stored by the algorithm is examined in turn here.

- The sweep plane is implemented as a two dimensional segment tree. Therefore its size is $O(n^2)$ as shown in Section 5.5.

- The number of subfaces is $n - 2$ per dimension as shown in Lemma 4.4.8, so these take $O(n)$ space.

- The size of the binary space partition tree is $O(\beta) = O(n^{3/2})$ as shown by Paterson and Yao [138].

- The number of blocks is $O(\beta) = O(n^{3/2})$ as shown by Theorem 4.4.19.

- There are $O(\beta n)$ neighbor relationships in the BSP decomposition of space, by Theorem 4.4.32, so this is the size of the neighbor graph which is built during preprocessing.

- Assuming duplicate events are deleted, the priority queue holds a maximum of $O(\beta)$ events, by Theorem 4.8.1. Each event might hold a one dimensional segment tree of size $O(n)$. Therefore the maximum size of the priority queue is $O(\beta n)$.

□

4.9 Constructing the Path

The algorithm, as we have discussed it thus far, only returns the number of links in a minimum link path. Of course, it would be useful to know what the actual path is, as well as its link-distance. We discuss how to obtain that here.

The precise description of an optimal path is built in two stages. First, a series of blocks through which the optimal path passes, and the rectangular regions, or stripes, through which the optimal path may enter the block is built. Second this series is used to define the series of line segments which will comprise the path. Section 4.9.1 describes how a the path can be stored in terms of the blocks through which it travels. Section 4.9.2 describes how to add another block to such a path description. Section 4.9.3 describes how to turn this list of blocks into a list of line segments.

4.9.1 Defining a Path in Terms of Blocks

During the course of our algorithm, we save events in a priority queue for later retrieval. Prior path information is stored in `OutgoingPath` events. (The other kinds of events, `EmptyBlock` and `ObstacleFace`, describe where a path may or may not go in the

future.) In order to be able reconstruct our path, we need to store the path thus far in `OutgoingPath` events.

We define the path here to be stored as series of path sections, where each section refers to a single block. Within each path section is stored a pointer to the block itself, the bounds of the rectangle through which it may have entered the block, and the number of turns the path can make within the block.

We define one path section variable as follows:

`PathSection =`

`Block:` *block*

`Dir:` *dir*

`Num:` $l,\ r,\ b,\ t$

`Integer:` *bends*

`PathSection:` *prev*

Here *block* is the block associated with this section of the path, *dir* is the direction of entry, l, r, b, t defines the bounds of the rectangle through which the path may enter the block, and *bends* is the number of bends the path may take within the block.

Note, it is important to keep the size of the rectangles, or stripes, at the time of block entry, not at the time of exit. This is because the exiting paths which the sweep plane picks up may subsequently be partially blocked as the sweep plane passes through obstacle faces, prior to entering another block.

Note also, that the sequence of blocks stored is not a comprehensive list of all blocks through which the path travels, but rather only those blocks in which the path makes a turn are stored. That is, blocks through which the path travels straight through without turning are not stored as path sections.

4.9.2 Adding a New Path Section

Here we describe the sequence of steps that are performed, in between the removal and insertion of an `OutgoingPath` event, in order to add a path section onto a saved path.

When an `OutgoingPath` event is removed from the queue, there is a one dimensional segment tree associated with that event. Some of the nodes of this tree may have path histories associated with them. The function INSERTSTRIPEDRECT or INSERTRECT is called in order to add the these paths to the sweep plane. The path histories associated with those nodes are also copied into the sweep plane.

Then, when an `EmptyBlock` event is processed, The function PROJECTRECT or QUERYRECT is called on the sweep plane. This is used to determine what paths may enter the block, and what new paths are inserted into the queue. Here we require two new functions QUERYRECT′ and PROJECTRECT′, designed to handle the path histories.

- QUERYRECT′ returns NULL if there is no path in the rectangle being queried, or returns a pointer to the node of the sweep plane which falls within that rectangle. If there is more than one such node, then any of the nodes is returned.

- PROJECTRECT′ projects the paths within a rectangle of the sweep plane onto an axis and returns the resulting one dimensional segment tree. The individual nodes of this segment tree are created with pointers to the sweep plane node that projects onto them. In case there is more than one sweep plane node projecting onto the same location, any such node is used.

From these operations we obtain a segment tree which describes a set of paths leaving the block associated with the `EmptyBlock` event. (Note, in the case of QUERYRECT′ a new empty segment tree is created, and the full range of the exit face of the returned block is inserted.) We then associate path histories with the individual nodes of the segment tree, before the tree is inserted back into the priority queue as an `OutgoingPath` event.

For each segment tree node, we keep a pointer to the sweep plane node from which it was derived. Using this sweep plane node we associate a new path section, P, with the segment tree node. We define $P.dir$ to be the direction of the sweep plane, $P.block$ to be the block associated with the `EmptyBlock` event, and $P.bends$ to be the number of bends taken within the block. Finally we define the entry rectangle associated with P to be the bounds of the sweep plane node.

4.9.3 Converting Path Sections into Line Segments

Once the path has been completely defined in terms of path sections, we would like to convert this into a valid series of line segments. Here we describe that conversion process.

Each path section contains a pointer to a block, and a rectangle on the face of the block through which the path may travel. Since the entry rectangles are taken straight from the sweep plane, it is possible that they may extend beyond the bounds of the block with which they are associated. Thus it is necessary to restrict the path to traveling through the intersection of the entry rectangle, and the entry face of the block.

Additionally, there may be entry rectangles and other blocks, later in the path, which further restrict how the path may travel into an earlier block. Therefore, we process the path sections in reverse order, beginning at the finish point.

Looking at the final path section, the intersection of the entry face into the block, and the entry rectangle of the path section defines an area that is reachable from the starting point by a uniform number of bends. So a path from anywhere in this rectangular area, having the number of bends within the block indicated by the path section, and finishing at the finish point can comprise part of a minimum link-distance path.

Processing all of the path sections like this in reverse order will give us an optimal path.

Chapter 5

Data Structures

Consider the following task. Rectilinearly partition d-dimensional space into two, not necessarily contiguous, sets of points. Each set has a boolean value, so the points in one set are considered empty and those in the other set are considered filled in.

Consider the problem of devising a data structure which supports the following dynamic operations on such a d-dimensional space:

- **[Insert]** Fill in all points within a specified hyperrectangle.
- **[Clear]** Clear all points within a specified hyperrectangle.
- **[Project]** Project the points within a specified hyperrectangle onto $(d-1)$ - dimensional space.
- **[Sweep]** Fill a specified hyperrectangle with the points from a $(d-1)$ - dimensional space, as if the lower-dimensional space had been "swept" across the hyperrectangle.
- **[Query]** Query a hyperrectangle to see if it contains any points.

In Chapter 4 this set of functions is required when $d = 2$ in order to perform rectilinear minimum link-distance queries in three dimensions [41].

Here, the implementation of a data structure supporting these operations when $d = 2$ is discussed. This implementation is adapted from the segment tree data structure, which was introduced by Bentley in 1977 [18]. Each operation requires amortized $O(n \log n)$ time, or less. Further, it seems likely that the operations can be implemented in actual $O(n \log n)$ time (removing the need for amortized analysis), as described in Section 7.2.1.

Segment trees are used extensively in computer graphics, and so the operations may also be useful in modeling scenarios where matter in the scene is created and removed in regions, rather than as a discrete set of objects [134].

The operations required in Chapter 4, and described in this chapter are:

- **[Insert]** Fill in all points within a specified rectangle.
- **[Clear]** Clear all points within a specified rectangle. (This operation does not mean the deletion of a previously inserted rectangle, which is supported by Bentley's segment trees.)
- **[Project]** Project all points which fall within the bounds of a given rectangle onto an axis, storing the result in a one dimensional data structure.
- **[Stripe]** Fill in all points within each of a set of rectangles. These rectangles span a given enclosing rectangle in one direction, and have widths or heights as specified by the set of intervals stored in a specified one dimensional data structure, thereby forming a set of "stripes" across the rectangle.
- **[Query]** Query a specified rectangle to see if it contains any points.

If this data structure were implemented naively, it might require $\Omega(n^2)$ time per operation, since after only n operations, there can be $\Omega(n^2)$ disconnected regions of points. A sequence which will cause this is the following:

1. Insert $\Theta(n)$ rectangles into the segment tree.

2. Clear $\Theta(n)$ rectangles in such a way that each previously inserted rectangle is split into $\Theta(n)$ pieces.

Here it is shown that all operations can be performed in amortized $O(n \log n)$ time per operation or better. In the case of the query and insert operations, these can be implemented to run in $O(\log^2 n)$ time, using the data structure described in this chapter.

Several extensions to this data structure are described toward the end of the section. In Section 5.6 the storing of an interval name at each node, instead of a boolean value, is discussed, thus recovering some of the original functionality of the segment tree. Section 7.2.2 describes methods for extending these results into higher dimensions, and Section 7.2.1 describes how the need for amortized analysis can be removed.

5.1 Background on the Segment Tree

In 1977 Klee posed the question "Can the total length of the union of n intervals be computed in less than $O(n \log n)$ steps?". He included with this question an algorithm achieving the bound [86]. Michael Fredman and Bruce Weide proved an $\Omega(n \log n)$ lower bound for the problem in 1978, thereby answering the question false and making Klee's method optimal [55]. However, this generated much interest in looking at the problem in two dimensions and higher. The problem quickly became known as "Klee's Rectangle Problem" in two dimensions, or "Klee's Measure Problem" in arbitrary dimensions.

In 1977 Jon Bentley wrote in an unpublished technical report an algorithm which solved Klee's Rectangle Problem in two dimensions in $O(n \log n)$ time, which by Fredman and Weide's lower bound, is also optimal. He further described that his method can be extended into higher dimensions, yielding a running time of $O(n^{d-1} \log n)$ for $d \geq 2$ [18].

Bentley's solution to the problem was later described in a publication of van Leeuwen and Wood. They further improved on Bentley's algorithm for Klee's Measure problem, solving it in $O(n^d - 1)$ for $d \geq 3$ [97]. This bound would stand until 1988, when Overmars and Yap improved it again with an algorithm running in $O(n^{d/2} \log n)$ time, for $d \geq 3$ [135].

In Bentley's report he introduced a novel data structure, in which various intervals were stored in the internal nodes of a perfectly balanced binary tree. A description of the Bentley's tree first appeared as a publication in 1980 [20]. These trees were designed to support the efficient insertion of intervals, and "stabbing" queries that return the set of the inserted intervals in which a query point lies, as well as possibly the deletion of a previously inserted interval.

In 1980 Edelsbrunner developed another data structure, designed to report all pairs of intersecting rectangles, which could be extended to report all intersections with a rectangle query. Details about this would be published in 1983 in two parts [46, 47].

In 1980 Edelsbrunner also wrote an extensive technical report about making data structures dynamic. He noticed that the queries of Bentley's tree structure were the inverse of another query, already widely known as the "range query". The range query asks for the set of all points falling within a query range, and it had been well known to be efficiently answerable with a data structure called the "range tree". Edelsbrunner's own intersection reporting queries were a generalization of both the range query and the stabbing query, using point-sized interval insertions or queries. He called his own intersection reporting data structure an "Interval Tree" and he called Bentley's data structure a "Segment Tree" [45].

Vaishnavi explained in 1982 how to extend the functionality of the segment tree to support the same operations efficiently in higher dimensions [157]. Several authors have described the use of such higher dimensional segment trees, especially for applications in computer graphics [33, 45, 48, 133, 158].

An overview of the segment tree can be found in a textbook of de Berg, van Kreveld, Overmars, and Schwarzkopf [23].

5.1.1 Implementation of Segment Trees

Segment trees store sets of intervals. Whether the intervals come from a domain of real numbers or integers, the segment tree breaks these intervals down into a set of discrete components, called "Elementary Intervals". Each elementary interval represents a smallest possible interval, and all intervals can be described as the union of a set of elementary intervals.

If the input is drawn from the real numbers, and the sorted set of interval endpoints is $\{p_1, p_2, \ldots, p_m\}$, then the elementary intervals are $(-\infty, p_1), [p_1, p_1], (p_1, p_2), [p_2, p_2], \ldots, [p_m, p_m], (p_m, +\infty)$ [23]. If the input is drawn from the integers, then set of elementary intervals can be similarly defined. Hereafter, we will simply refer all intervals in terms of the sorted positions of the n elementary intervals which comprise them, from among positions $\mu_1, \ldots, \mu_n$, where $\mu_1 = (-\infty, p_1), \mu_2 = [p_1, p_1], \mu_3 = (p_1, p_2), \ldots, \mu_n = (p_m, +\infty)$. So, the entire range covered by the segment tree is $[\mu_1, \mu_n]$.

A segment tree covering the range $[\mu_1, \mu_n]$ is implemented as a balanced binary tree of height $\lceil \log n \rceil$. Each node of the tree represents some subrange of $[\mu_1, \mu_n]$. The root of the tree represents the entire range $[\mu_1, \mu_n]$, its two children represent $[\mu_1, \mu_{\lfloor \frac{n}{2} \rfloor}]$ and $[\mu_{\lfloor \frac{n}{2} \rfloor + 1}, \mu_n]$, and the leaves of the tree represent the smallest elementary ranges, $[\mu_1, \mu_1]$, $[\mu_2, \mu_2]$, etc. Any non-leaf node represents the union of the ranges of its children. See Figure 5.1.

Unlike in interval trees and range trees[45, 114], where a single inserted interval is stored at a constant number of nodes, an interval in a segment tree is stored at $O(\log n)$ of the nodes in the tree. The particular subset of nodes chosen is unique for a given range, and

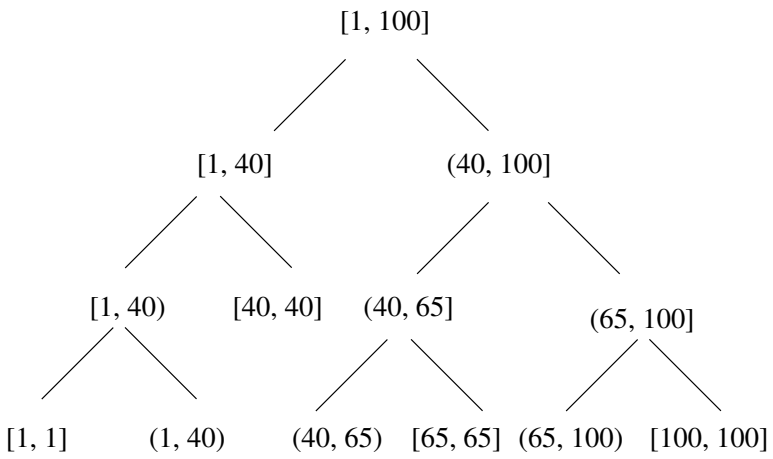


Figure 5.1: An example of a segment tree over the range $[1, 100]$, and containing the points $1, 40, 65$, and 100

so we will refer to those nodes as the *canonical nodes* representing that range.

To determine the set of canonical nodes representing a range $[\mu_l, \mu_r]$, begin with the set of leaves which represent the ranges $[\mu_l, \mu_l]$, $[\mu_{l+1}, \mu_{l+1}]$, $\ldots$, $[\mu_r, \mu_r]$. Next, look for a pair of siblings from this set. If they exist, remove these siblings, and insert their parent. Continue to perform this operation until there is no pair of siblings in the set. The resulting set of nodes is the canonical set representing that range.

This choice of nodes is made so that the union of the ranges represented by all of these nodes is equivalent to the entire range being represented. Further, the intersection of the ranges represented by any two of the canonical nodes is empty, and there are $O(\log n)$ nodes in the set.

Since more than one range can be stored in the same segment tree, it is possible that the same node will be among the set of canonical nodes representing more than one inserted range. Each node therefore keeps a list of all the inserted ranges for which it is a canonical node.

The functionality of segment trees can be extended to operate in two dimensions. This is done by attaching a one dimensional segment tree to each node of a primary tree. The

data structure then supports the insertion of rectangles, the deletion of previously inserted rectangles, and the query of two dimensional data points. Insertion and deletion then run in $O(\log^2 n)$ time, and stabbing queries run in $O(\log n)$ time [157].

Note that a one dimensional segment tree storing n intervals requires $O(n \log n)$ space, since each interval is stored at most a constant number of times in each of $O(\log n)$ levels of a balanced binary tree. Likewise, a two dimensional segment tree storing n rectangles requires $O(n \log^2 n)$ storage space, since each interval is stored at most a constant number of times in each level of each of $O(\log n)$ trees [134].

5.2 One Dimensional Operations

Here we will describe a new variation on the traditional segment tree. This variation supports the insertion of intervals and the query of intervals. In this sense it behaves similarly to an interval tree which also supports these operations.

The primary difference is that this kind of segment tree does not differentiate between different intervals after they have been inserted, but instead maintains the union of all inserted intervals. In exchange for this sacrifice, the tree supports a new clear operation, defined in Section 5.2.3. Furthermore, two dimensional versions of this data structure support the full set of operations described above in Section 5. Some may find this tree theoretically elegant, requiring just the addition of two bits at each node of a segment tree.

Let's begin by defining a node of this kind of segment tree. At each node we keep a boolean variable $data$ which, if set, indicates that the entire range represented by the node is covered. We will sometimes refer to this variable as the *data bit* associated with this node of the tree.

By storing only a boolean value here, the ability to distinguish between different intervals is lost. Instead this kind of node stores TRUE to indicate that the entire range

represented by the node is covered by one or more inserted intervals. As a result, a query operation can only report whether the point queried is within some inserted interval or not. It cannot report the specific intervals within which it lies. A discussion on how to partially recover this functionality appears in section 5.6.

A node thus keeps the following data:

- Pointers to its left and right children and to its parent within the tree
- A bit $data$ indicating that the entire range represented by this node is covered by one or more inserted intervals. It is not necessary to keep the list of the intervals, as is done with Bentley's segment trees, since only the union of all intervals is maintained.
- A bit $subtreedata$ determining if the data bit is set in any node in the entire subtree rooted at this node.
- The range $[\mu_l, \mu_r]$ which this node represents.

The following data structure describes a node of our new kind of one dimensional segment tree:

```
SegTreeNode =
   SegTreeNode:  left, right, parent
   Boolean:  data, subtreedata
   Num:  l, r
```

The node is augmented with another boolean variable $subtreedata$. This variable is TRUE if any node in the entire subtree rooted at this node has its data bit set to TRUE. If set to FALSE, this indicates that no part of the range represented by the node is covered.

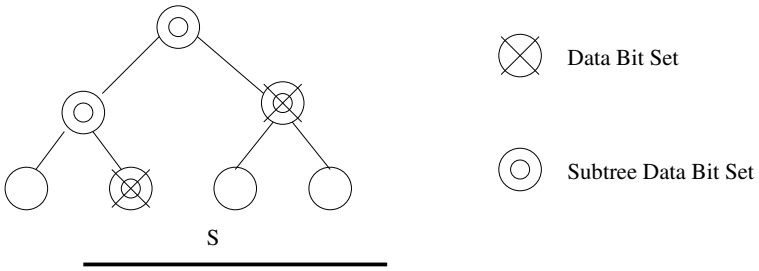


Figure 5.2: The data bits and subtree data bits set after the insertion of segment S

It is used to avoid searching an entire subtree for any TRUE data bit. This variable will sometimes be referred to as the *subtreedata bit*. See Figure 5.2.

We define the following function to maintain the value of $subtreedata$:

Algorithm 11 MAINTAINSUBTREEDATA(T)

Given: A tree node, T, from a one dimensional segment tree

1: **if** (T is a leaf) **then**
2: $T.subtreedata \leftarrow T.data$
3: **else**
4: $T.subtreedata \leftarrow T.data$ or
$T.left.subtreedata$ or
$T.right.subtreedata$

This function runs in constant time.

5.2.1 Determining Canonical Nodes

As mentioned in Section 5.1, for any given range there is a unique set of non-overlapping nodes which represent that range. These are referred to as the canonical nodes of that range. Thus, for a given range, we can classify each node of the tree as either a canonical node, an ancestor of a canonical node, a descendant of a canonical node, or none of these (for example, a sibling of a canonical node would fall into this last category).

The following function returns TRUE if the node is either a canonical node, an ancestor of a canonical node, or a descendant of a canonical node representing the range $[\mu_l, \mu_r]$.

Algorithm 12 IsCanonicalOrAncOrDec(T, l, r)

Given: A tree node, T, from a one dimensional segment tree and a range $[\mu_l, \mu_r]$

Return: TRUE if T is a canonical node representing the range, or an ancestor or descendant of such a canonical node

1: **return** $(l < T.r)$ or $(T.l < r)$

The following function returns TRUE if the node is either a canonical node or a descendant of a canonical node.

Algorithm 13 IsCanonicalOrDec(T, l, r)

Given: A tree node, T, from a one dimensional segment tree and a range $[\mu_l, \mu_r]$

Return: TRUE if T is a canonical node representing the range, or a descendant of such a canonical node

1: **return** $(r \leq T.r)$ and $(T.l \leq l)$

By running a combination of the above two functions on a node and its parent, one can properly categorize a node as either a canonical node, an ancestor of a canonical node, a descendant of a canonical node, or none of these.

For a given range $[\mu_l, \mu_r]$ the following functions determine whether a node is a canonical node representing that range, an ancestor of such a canonical node, a descendant of a canonical node, or if the node does not overlap the range at all. Each of these takes constant time.

- Boolean:IsCanonical(T, l, r)

  Return TRUE if segment tree node T is among the set of canonical nodes representing the range $[\mu_l, \mu_r]$.

- Boolean:IsCanonicalDec(T, l, r)

  Return TRUE if segment tree node T is a descendant of a canonical node representing the range $[\mu_l, \mu_r]$.

- Boolean:IsCanonicalAnc(T, l, r)

Return TRUE if segment tree node T is an ancestor of a canonical node representing the range $[\mu_l, \mu_r]$.

- Boolean:NOTWITHINRANGE(T, l, r)

  Return TRUE if T is NULL. Return TRUE if segment tree node T is neither a canonical node, nor a descendant nor an ancestor of a canonical node representing the range $[\mu_l, \mu_r]$.

5.2.2 Splitting and Joining the Data

It will often be helpful to push the data bit stored at a node down onto its two children. The total range represented after this operation is unchanged, however we can then work with each of the two halves individually. This can be a precursor to a Clear operation which only needs to be performed on part of the range represented by the node. We call this operation "Splitting" the data.

Likewise, if two siblings have their data bit set, it may be more efficient to set their parent instead. After an insert operation this can be used to clean up the data in a tree.

We define these two functions:

Algorithm 14 SPLITDATA(T)

Given: A tree node, T, from a one dimensional segment tree

1: **if** ((T is not a leaf) and ($T.data =$ TRUE)) **then**
2: $T.left.data \leftarrow$ TRUE
3: $T.right.data \leftarrow$ TRUE
4: $T.left.subtreedata \leftarrow$ TRUE
5: $T.right.subtreedata \leftarrow$ TRUE
6: $T.data \leftarrow$ FALSE

These maintenance functions update the values of both $data$ and $subtreedata$ within the node T and its two children. This is only done if the data supports the operation (e.g. SPLITDATA does nothing if $T.data$ is FALSE).

Algorithm 15 JOINDATA(T)

Given: A tree node, T, from a one dimensional segment tree

1: **if** ((T is not a leaf) and ($T.left.data =$ TRUE) and ($T.right.data =$ TRUE)) **then**
2: $T.left.data \leftarrow$ FALSE
3: $T.right.data \leftarrow$ FALSE
4: $T.left.subtreedata \leftarrow$ FALSE
5: $T.right.subtreedata \leftarrow$ FALSE
6: $T.subtreedata \leftarrow$ TRUE
7: $T.data \leftarrow$ TRUE

5.2.3 Operations on One Dimensional Segment Trees

Here we will define the following operations on one dimensional segment trees.

- INSERTRANGE(T, l, r)

  Inserts the range $[\mu_l, \mu_r]$ into the segment tree T.

- CLEARRANGE(T, l, r)

  Clears from segment tree T the set of all points in the range $[\mu_l, \mu_r]$.

- Boolean: QUERYRANGE(T, l, r)

  Queries segment tree T, returning TRUE if there are any points within the range $[\mu_l, \mu_r]$.

The running times of these operations are listed in Table 5.1

INSERTRANGE	$O(\log n + D)$
CLEARRANGE	$O(\log n + D)$
QUERYRANGE	$O(\log n)$

Table 5.1: The running times of our one dimensional segment trees operations. Here D is the number of subtreedata bits cleared by the operation.

Inserting a Range

Inserting a range is done by setting every canonical node in the representation of that range to TRUE. This may create some redundancy, with ancestor and descendant pairs both being marked TRUE, so we remove all such marks in the subtree of both children of every canonical node.

We must also maintain the boolean variable $subtreedata$ and join together any siblings which are marked TRUE.

Algorithm 16 INSERTRANGE(T, l, r)

Given: A tree node, T, from a one dimensional segment tree and a range $[\mu_l, \mu_r]$

```
 1: if ( (T = NULL) or (NOTWITHINRANGE(T, l, r)) ) then
 2:    return
 3: if ( ISCANONICAL(T, l, r) ) then
 4:    T.data ← TRUE
 5: if ( ISCANONICALDEC(T, l, r) ) then
 6:    T.data ← FALSE
 7:    if ( T.subtreedata = FALSE ) then
 8:       return
 9: INSERTRANGE(T.left, l, r)
10: INSERTRANGE(T.right, l, r)
11: MAINTAINSUBTREEDATA(T)
12: JOINDATA(T)
```

Theorem 5.2.1 *The total time to insert a range into a segment tree is $O(\log n + D)$ where D is the number of subtreedata bits cleared in the tree by calls to* MAINTAINSUBTREEDATA. *Note, D may be as large as $\Theta(n)$.*

Proof. This function only considers canonical nodes, their ancestors and descendants, and the children of ancestors. All operations at a node, including MAINTAINSUBTREEDATA take constant time. There are $O(\log n)$ nodes in the set of canonical nodes representing any range. Furthermore, there are $O(\log n)$ nodes in the set of all ancestors of these canonical nodes, as well as the set of all children of ancestors. Among the set of descendants of

canonical nodes, only those nodes whose subtreedata bit is set to TRUE and their children are considered, and there are $O(D)$ of these. □

Clearing a Range

Clearing out a range is done by clearing all data in the entire subtree rooted at every canonical node. Additionally, any ancestor of a canonical node which is set to TRUE must be split into an equivalent set of descendants. The ancestor itself is set to FALSE. Its descendant is set to FALSE if it is a canonical node, set to TRUE if it falls outside the range being cleared, or recursed upon if it is still an ancestor of a canonical node. Processing the ancestors top down from the root, splitting each onto its two children, and recursing, will perform this operation properly.

The following CLEARRANGE function clears all data within the indicated range in the one dimensional segment tree.

Algorithm 17 CLEARRANGE(T, l, r)

Given: A tree node, T, from a one dimensional segment tree and a range $[\mu_l, \mu_r]$

```
 1: if ( (T = NULL) or (NOTWITHINRANGE(T, l, r)) ) then
 2:    return
 3: if ( T.subtreedata = FALSE ) then
 4:    return
 5: if ( ISCANONICALANC(T, l, r) ) then
 6:    SPLITDATA(T)
 7: if ( ISCANONICAL(T, l, r) or ISCANONICALDEC(T, l, r) ) then
 8:    T.data ← FALSE
 9: CLEARRANGE(T.left, l, r)
10: CLEARRANGE(T.right, l, r)
11: MAINTAINSUBTREEDATA(T)
```

Theorem 5.2.2 *The ClearRange function runs in time $O(\log n + D)$, where D is the number of subtreedata bits cleared in the tree by calls to* MAINTAINSUBTREEDATA. *Note, D may be as large as $\Theta(n)$.*

Proof. This function only considers canonical nodes, their ancestors and descendants, and the children of ancestors. All operations at a node, including SPLITDATA and MAINTAINSUBTREEDATA take constant time. There are $O(\log n)$ nodes in the set of canonical nodes representing any range. Furthermore, there are $O(\log n)$ nodes in the set of all ancestors of these canonical nodes, as well as their children. Among the set of descendants of canonical nodes, only those nodes whose subtreedata bit is set to TRUE and their children are considered, and there are $O(D)$ of these. □

Querying a Range

Querying a segment tree traditionally has meant querying a data point, and returning the set of all ranges within which that data point was contained.

For our purposes, however, we instead want to ask if there is any data within a range. Therefore, we define a query function which queries a range within the segment tree. The query function returns TRUE if there is a data bit set anywhere within the specified range.

This is performed by reading the data bit on all ancestors of canonical nodes, and by reading the subtreedata bit on all canonical nodes.

Algorithm 18 QUERYRANGE(T, l, r)

Given: A tree node, T, from a one dimensional segment tree and a range $[\mu_l, \mu_r]$
Return: `TRUE` if any data in segment tree T is stored within the range $[\mu_l, \mu_r]$, or `FALSE` otherwise.

```
1: if ( (T = NULL) or (NOTWITHINRANGE(T, l, r)) ) then
2:     return FALSE
3: if ( T.subtreedata = FALSE ) then
4:     return FALSE
5: if ( T.data = TRUE ) then
6:     return TRUE
7: if ( ISCANONICAL(T, l, r) ) then
8:     return T.subtreedata
9: return QUERYRANGE(T.left, l, r) or
   QUERYRANGE(T.right, l, r)
```

Theorem 5.2.3 *The QueryRange function runs in $O(\log n)$ time.*

Proof. Only canonical nodes, their ancestors, and the children of ancestors are considered in this function. □

5.2.4 Further Operations on One Dimensional Segment Trees

Here we define some additional operations on one dimensional segment trees. These will be used as helper functions within some of the operations on two dimensional segment trees, defined later in section 5.3.

- COPYTREE(T, S)

  Copy the contents of the segment tree S into the segment tree T, overwriting it.

- UNIONTREES(T, S)

  Union all the ranges in the two trees S and T together, overwriting T with the result. The subtreedata bits are updated as necessary.

- TRUNCATETREE(T, l, r)

  Truncate the intervals in T, removing all data which falls outside the range $[\mu_l, \mu_r]$.

- NEWSEGTREENODE(l, r)

  Return a new segment tree node representing the range $[\mu_l, \mu_r]$, setting its data bit to FALSE.

- NEWSEGTREE(l, r)

  Create a new segment tree spanning the range $[\mu_l, \mu_r]$. This will become a subtree of the larger segment tree spanning $[\mu_1, \mu_n]$. Set all data in all nodes of this tree to FALSE. Return a pointer to the root of this tree.

Table 5.2 lists the running times of the above functions.

COPYTREE	$O(n)$
UNIONTREES	$O(n)$
TRUNCATETREE	$O(n)$
NEWSEGTREENODE	$O(1)$
NEWSEGTREE	$O(r-l)$

Table 5.2: The running times of some utility functions on our one dimensional segment trees.

5.3 Two Dimensional Operations

It is possible to use segment trees to represent data in two dimensions. Such a tree allows the insertion and deletion of rectangles, rather than of ranges.

Beginning with a one dimensional segment tree, T, a second dimension is added by attaching a one dimensional segment tree to each node of T.

The following data structure describes a node in the top level tree of a two dimensional segment tree.

```
SegTreeNode2D =
   SegTreeNode2D: left, right, parent
   SegTreeNode:   perptree, projtree
   Num:   b, t
```

The segment tree variable *perptree* is a pointer to the root of a second level tree. This second level tree is a one dimensional segment tree, as defined as in section 5.2.

We will augment each of these nodes with a pointer to an additional one dimensional segment tree. This data structure, which we call a "Projection Tree", stores the projection of all *perptree* variables attached to any first level tree node which is a descendant of this node. The variable *projtree* is a pointer to the root of the projection tree.

The value of $T.projtree$ is maintained as the union of $T.perptree$ and each of the projection trees of the two children of T. Algorithm 19 maintains the projection tree of a node:

Algorithm 19 MAINTAINPROJTREE$(XYTree)$

Given: A tree node, $XYTree$, from a two dimensional segment tree

```
1: COPYTREE(XYTree.projtree, XYTree.perptree)
2: if ( XYTree is not a leaf ) then
3:     UNIONTREES(XYTree.projtree, XYTree.left.projtree)
4:     UNIONTREES(XYTree.projtree, XYTree.right.projtree)
```

This maintenance function takes $O(n)$ time.

5.3.1 Operations on Two Dimensional Segment Trees

Here we will define the following operations on two dimensional segment trees.

- INSERTRECT$(XYTree, l, r, b, t)$

  Insert the rectangle $[\mu_l, \mu_r] \times [\mu_b, \mu_t]$ into the two dimensional segment tree $XYTree$.

- INSERTSTRIPEDRECT$(XYTree,\ Tree, l, r, b, t, sDir)$

  Stripe the rectangle $[\mu_l, \mu_r] \times [\mu_b, \mu_t]$ in the two dimensional segment tree $XYTree$ according to the contents of the one dimensional segment tree $Tree$. The direction of striping is either x or y as indicated by $sDir$.

- CLEARRECT$(XYTree, l, r, b, t)$

  Clear all points from the two dimensional segment tree $XYTree$ which lie within the rectangle $[\mu_l, \mu_r] \times [\mu_b, \mu_t]$.

- QUERYRECT$(XYTree, l, r, b, t)$

  Query the two dimensional segment tree $XYTree$, returning TRUE if $XYTree$ contains any points within the rectangle $[\mu_l, \mu_r] \times [\mu_b, \mu_t]$.

- PROJECTRECT$(XYTree, tree, l, r, b, t, pDir)$

  Project the contents of the rectangle $[\mu_l, \mu_r] \times [\mu_b, \mu_t]$ within the two dimensional segment tree $XYTree$ onto one of the axes, storing the result in the one dimensional segment tree $Tree$. The choice of axis, either x or y, is indicated by $sDir$.

The running times of the operations are listed in Table 5.3.

INSERTRECT	$O(\log^2 n + D)$
INSERTSTRIPEDRECT	$O(n \log n + D)$
CLEARRECT	$O(n \log n + D)$
QUERYRECT	$O(\log^2 n)$
PROJECTRECT	$O(n \log n)$

Table 5.3: The running times of our two dimensional segment tree operations. Again, D is the number of subtreedata bits cleared by the operation.

Inserting a Rectangle

Here we describe how to insert a rectangle into a two-dimensional segment tree. We define the range $[\mu_l, \mu_r]$ to span between the leftmost and rightmost point of the rectangle in the x dimension, and $[\mu_b, \mu_t]$ to span between the bottommost and topmost point in the rectangle in the y dimension. The top level of the two-dimensional segment tree runs in the y dimension, so a top level node contains the variables b and t.

In order to insert a rectangle $[\mu_l, \mu_r] \times [\mu_b, \mu_t]$ into a segment tree, we determine which are the canonical nodes in the segment tree representing the range $[\mu_b, \mu_t]$, spanning the y dimension. Call these nodes $YNodes$. For each node $ynode_i \in YNodes$, insert the range $[\mu_l, \mu_r]$ into its second level tree, $perptree$.

Theorem 5.3.1 *The total time to insert a rectangle into a two dimensional segment tree is $O(\log^2 n + D)$, where D is the number of subtreedata bits cleared in the tree. Note that D may be as large as $\Theta(n^2)$.*

Algorithm 20 INSERTRECT$(XYTree, l, r, b, t)$

Given: A tree node, $XYTree$, from a two dimensional segment tree, and the bounds of a rectangle $[\mu_l, \mu_r] \times [\mu_b, \mu_t]$

```
 1: if ( (XYTree = NULL) or (NOTWITHINRANGE(XYTree, b, t)) ) then
 2:     return
 3: if ( ISCANONICALDEC(XYTree, b, t) ) then
 4:     return
 5: if ( ISCANONICALANC(XYTree, b, t) ) then
 6:     INSERTRANGE(XYTree.projtree, l, r)
 7: if ( ISCANONICAL(XYTree, b, t) ) then
 8:     INSERTRANGE(XYTree.perptree, l, r)
 9:     INSERTRANGE(XYTree.projtree, l, r)
10:     return
11: INSERTRECT(XYTree.left, l, r, b, t)
12: INSERTRECT(XYTree.right, l, r, b, t)
```

Proof. Within the top level tree, only canonical nodes, their ancestors, and the children of both of these are ever considered, and there are only $O(\log n)$ of each. The INSERTRANGE function is called twice on each canonical node and once on each ancestor. Each invocation takes $O(\log n)$ time plus time proportional to the number of nodes whose subtreedata bit is cleared. □

Inserting a Striped Rectangle

In this section we describe how to stripe the area inside a rectangle within an existing two dimensional segment tree. The contents of a one dimensional segment tree are used to determine variations along one axis within the rectangle. The stripes extend the full length of the rectangle along the other axis.

The function differs somewhat depending on whether the rectangle is striped in the x dimension or in the y dimension. This implementation assumes there are no stripes outside the rectangle limits in the input segment tree, and so one of the dimensions of the rectangle is not given. If the input tree did have data outside these limits, it would be an $O(n)$

operation to truncate the tree, prior to calling this function.

The function INSERTSTRIPEDRECT calls one of either INSERTSTRIPEDRECTX or INSERTSTRIPEDRECTY, depending on the striping direction.

Algorithm 21 INSERTSTRIPEDRECTX$(XYTree,\ YTree, l, r)$

Given: A tree node, $XYTree$, from a two dimensional segment tree, a tree node, $YTree$, from a one dimensional segment tree, and the bounds of a rectangle in the x dimension $[\mu_l, \mu_r]$

1: **if** (($XYTree =$ `NULL`) or ($YTree =$ `NULL`) or (not $YTree.subtreedata$)) **then**
2: **return**
3: INSERTRANGE$(XYTree.projtree, l, r)$
4: **if** ($YTree.data$) **then**
5: INSERTRANGE$(XYTree.perptree, l, r)$
6: INSERTSTRIPEDRECTX$(XYTree.left,\ YTree.left, l, r)$
7: INSERTSTRIPEDRECTX$(XYTree.right,\ YTree.right, l, r)$

Theorem 5.3.2 *The time to call the InsertStripedRectX function is $O(n \log n + D)$, where D is the number of subtreedata bits cleared in the tree. Note that D may be as large as $\Theta(n^2)$.*

Proof. In InsertStripedRectX, the InsertRange function may be called on each node of the top level tree. There are $O(n)$ of these nodes, but each call will only take $O(\log n + D_i)$, where D_i is the number of subtreedata bits cleared calling InsertRange on node i. □

Theorem 5.3.3 *The time to call the InsertStripedRectY function is $O(n \log n)$.*

Proof. InsertStripedRectY calls UnionTrees twice for each canonical node and once for each ancestor of a canonical node. There are $O(\log n)$ of these and each call can take $O(n)$. □

Corollary 5.3.4 *The time to insert a striped rectangle into a two dimensional segment tree is $O(n \log n + D)$, where D is the number of nodes whose subtreedata bit is cleared.*

Algorithm 22 INSERTSTRIPEDRECTY$(XYTree, XTree, b, t)$

Given: A tree node, $XYTree$, from a two dimensional segment tree, A tree node, $XTree$, from a one dimensional segment tree, and the bounds of a rectangle in the y dimension $[\mu_b, \mu_t]$

1: **if** (($XYTree =$ NULL) or ($XTree =$ NULL) or (NOTWITHINRANGE$(XYTree, b, t)$)) **then**
2: **return**
3: **if** (ISCANONICALDEC$(XYTree, b, t)$) **then**
4: **return**
5: UNIONTREES$(XYTree.projtree, XTree)$
6: **if** (ISCANONICAL$(XYTree, b, t)$) **then**
7: UNIONTREES$(XYTree.perptree, XTree)$
8: INSERTSTRIPEDRECTY$(XYTree.left, XTree, b, t)$
9: INSERTSTRIPEDRECTY$(XYTree.right, XTree, b, t)$

Clearing a Rectangle

Clearing out a rectangle is done by calling ClearRange on all nodes within the entire subtree rooted at every canonical node.

Preceding this, the data in the second level tree of any ancestor of a canonical node may need to be split onto an equivalent set of descendants, before being cleared out itself. Processing the ancestor nodes top down beginning at the root, unioning each onto its two children, and clearing it entirely, will perform this operation correctly.

Theorem 5.3.5 *The* CLEARRECT *function runs in time* $O(n \log n + D)$ *where* D *is the number of subtreedata bits cleared in the tree. Note that* D *may be as large as* $\Theta(n^2)$.

Proof. For each canonical node and descendant of a canonical node we do $O(\log n + D_i)$ work in ClearRange, where D_i is the number of subtreedata bits cleared by calling ClearRange on node i, and there are $O(n)$ of these nodes. For each ancestor we spend $O(n)$ time in calls to UnionTree, plus $O(\log n + D_i)$ time in ClearRange, and $O(n)$ time in MaintainProjTree. There are $O(\log n)$ of these nodes. Since $D = \sum D_i$, the total time for this is $O(n \log n + D)$. □

Algorithm 23 CLEARRECT$(XYTree, l, r, b, t)$

Given: A tree node, $XYTree$, from a two dimensional segment tree, and the bounds of a rectangle $[\mu_l, \mu_r] \times [\mu_b, \mu_t]$

```
1: if ( (XYTree = NULL) or (NOTWITHINRANGE(XYTree, b, t)) ) then
2:    return
3: if ( ISCANONICALANC(XYTree, b, t) ) then
4:    UNIONTREES(XYTree.left.perptree, XYTree.perptree)
5:    UNIONTREES(XYTree.left.projtree, XYTree.perptree)
6:    UNIONTREES(XYTree.right.perptree, XYTree.perptree)
7:    UNIONTREES(XYTree.right.projtree, XYTree.perptree)
8:    CLEARRANGE(XYTree.perptree, XYTree.perptree.l, XYTree.perptree.r)
9: else
10:   CLEARRANGE(XYTree.perptree, l, r)
11:   CLEARRANGE(XYTree.projtree, l, r)
12: CLEARRECT(XYTree.left, l, r, t, b)
13: CLEARRECT(XYTree.right, l, r, t, b)
14: if ( ISCANONICALANC(XYTree, b, t) ) then
15:   MAINTAINPROJTREE(XYTree)
```

Querying a Rectangle

Querying a rectangle involves looking at all ancestors of all canonical nodes. For each of these the QUERYRANGE function is called on the associated second level tree. Then QUERYRANGE is called on the projection tree of each canonical node.

Algorithm 24 QUERYRECT$(XYTree, l, r, b, t)$

Given: A tree node, $XYTree$, from a two dimensional segment tree, and the bounds of a rectangle $[\mu_l, \mu_r] \times [\mu_b, \mu_t]$

```
1: if ( (XYTree = NULL) or (NOTWITHINRANGE(XYTree, b, t)) ) then
2:    return FALSE
3: if ( ISCANONICAL(XYTree, b, t) ) then
4:    return QUERYRANGE(XYTree.projtree, l, r)
5: if ( QUERYRANGE(XYTree.perptree, l, r) ) then
6:    return TRUE
7: return QUERYRECT(XYTree.left, l, r, b, t) or QUERYRECT(XYTree.right, l, r, b, t)
```

Theorem 5.3.6 *The total time to query a rectangle in a two dimensional segment tree is* $O(\log^2 n)$.

Proof. The QueryRange function may be called for every canonical node, as well as every ancestor of a canonical node. There are $O(\log n)$ of these and each call takes $O(\log n)$. □

Projecting a Rectangle onto an Axis

Here we describe how to project the data in a rectangle within a two dimensional segment tree onto one of its two axes, and to build the one dimensional segment tree representing that projection. We assume that the data to be projected may extend beyond the boundaries of the rectangle. Thus the result of this operation is the projection of the data after it has been cropped to fit the rectangle. Again the implementation will differ somewhat, depending on which axis we are projecting onto.

First, let's handle the case where the axis we are projecting onto is the y-axis, which runs parallel to the top level of the segment tree. In order to project a rectangle $[\mu_l, \mu_r] \times [\mu_b, \mu_t]$, we must consider the canonical nodes, $YNodes$, in the top level of the segment tree representing the range $[\mu_b, \mu_t]$.

For each canonical node, $ynode_i \in YNodes$, we determine if anything projects onto the range represented by that node. The data in the perpendicularly aligned second level tree attached to $ynode_i$ can project onto this range. Any data within the range $[\mu_l, \mu_r]$ of the second level tree will project onto the entire range represented by $ynode_i$. We call call the QUERYRANGE function to determine this.

Furthermore, the data in any ancestor of $ynode_i$ can also project onto this range. Since the projection is cropped to fit the range $[\mu_b, \mu_t]$, the projection of any ancestor of a canonical node is equivalent to the projection of the canonical node itself.

If neither the canonical node, $ynode_i$, nor any of its ancestors projects onto the range, then the descendants of $ynode_i$ are considered. A descendant will project onto only part of the range represented by $ynode_i$.

Algorithm 25 performs the projection onto the y-axis:

Algorithm 25 PROJECTRECTY$(XYTree, l, r, b, t)$

Given: A tree node, $XYTree$, from a two dimensional segment tree, and the bounds of a rectangle $[\mu_l, \mu_r] \times [\mu_b, \mu_t]$

Return: A one dimensional segment tree containing the projection of the rectangle $[\mu_l, \mu_r] \times [\mu_b, \mu_t]$ onto the y-axis

```
1: if ( (XYTree = NULL) or (NOTWITHINRANGE(XYTree, b, t)) ) then
2:    return NEWSEGTREE(XYTree.b, XYTree.t)
3: if ( QUERYRANGE(XYTree.perptree, l, r) ) then
4:    YTree ← NEWSEGTREE(XYTree.b, XYTree.t)
5:    INSERTRANGE(YTree, b, t)
6:    return YTree
7: YTree ← NEWSEGTREENODE(XYTree.b, XYTree.t)
8: YTree.left ← PROJECTRECTY(XYTree.left, l, r, b, t)
9: YTree.right ← PROJECTRECTY(XYTree.right, l, r, b, t)
10: MAINTAINSUBTREEDATA(YTree)
11: JOINDATA(YTree)
12: return YTree
```

Theorem 5.3.7 *The ProjectRectY function runs in $O(n \log n)$ time.*

Proof. QueryRange and InsertRange are called at most once per node in the top level tree. The running time of QueryRange is $O(\log n)$, and the running time of InsertRange is $O(\log n + D_i)$, where D_i is the number of subtreedata bits cleared by calling InsertRange on node i. InsertRange is only called on an empty tree, though, so $D_i = 0$. There are $O(n)$ of these nodes, so $O(n \log n)$ time is spent in QueryRange and InsertRange.

Additionally, the time spent in a call to NewSegTree is proportional to the size of the tree being created. All trees created in this manner eventually become a subtree of the whole tree returned at the end. This tree has size $O(n)$, so there is $O(n)$ time spent in calls to NewSegTree. □

Next, let's consider the projection onto the other axis. In this case, the axis of projection runs parallel to that represented by the second level of the tree.

Determining the projection onto the x-axis can be done by taking the union of the projection trees of the canonical nodes and the second level trees of their ancestors. The resulting segment tree must be pruned to the proper dimensions.

Algorithm 26 PROJECTRECTX$(XYTree, l, r, b, t)$

Given: A tree node, $XYTree$, from a two dimensional segment tree, and the bounds of a rectangle $[\mu_l, \mu_r] \times [\mu_b, \mu_t]$

Return: A one dimensional segment tree containing the projection of the rectangle $[\mu_l, \mu_r] \times [\mu_b, \mu_t]$ onto the x-axis

```
1: if ( (XYTree = NULL) or (NOTWITHINRANGE(XYTree, b, t)) ) then
2:    return NEWSEGTREE(XYTree.perptree.l, XYTree.perptree.r)
3: XTree ← NEWSEGTREE(XYTree.perptree.l, XYTree.perptree.r)
4: if ( ISCANONICAL(XYTree, b, t) ) then
5:    UNIONTREES(XTree, XYTree.projtree)
6:    TRUNCATETREE(XTree, l, r)
7:    return XTree
8: UNIONTREES(XTree, XYTree.perptree)
9: UNIONTREES(XTree, PROJECTRECTX(XYTree.left, l, r, b, t)
10: UNIONTREES(XTree, PROJECTRECTX(XYTree.right, l, r, b, t))
11: TRUNCATETREE(XTree, l, r)
12: return XTree
```

Theorem 5.3.8 *The ProjectRectX function runs in $O(n \log n)$ time.*

Proof. We only look at the projection trees of canonical nodes and the second level trees of their ancestors. Since there can be $O(\log n)$ of these, the total time to union them together is $O(n \log n)$. NewSegTree and TruncateTree are also called once for each of these nodes, and each call takes $O(n)$. □

Corollary 5.3.9 *It takes $O(n \log n)$ to project a rectangle onto an axis.*

5.4 Runtime Analysis

In this chapter we have shown how to implement the pointwise boolean insertion and deletion of arbitrary rectangles in a two dimensional segment tree. We have also shown how to project the data within any rectangle of a two dimensional segment tree onto an axis, as well as how to stripe the data from an axis into a rectangle, and how to query a rectangle for data.

If these operations were implemented naively, certain combinations of operations could take $\Omega(n^2)$ time per operation, where n is the number of rectangles. This is due to the possibility that $\Omega(n)$ inserted rectangles could each be fragmented into $\Omega(n)$ pieces by deletion operations, and the resulting $\Omega(n^2)$ pieces projected repeatedly.

A main result of this chapter is that, beginning with an empty two dimensional segment tree, our implementation performs any sequence of these operations at an amortized cost of $O(n \log n)$ per operation. We prove that theorem here:

Theorem 5.4.1 *Beginning with an empty two dimensional segment tree, consider any sequence of n of the following operations:*

1. INSERTRECT$(XYTree, l, r, b, t)$
2. CLEARRECT$(XYTree, l, r, b, t)$
3. `SegTreeNode:` PROJECTRECT$(XYTree, Tree, l, r, b, t, dir)$
4. INSERTSTRIPEDRECT$(XYTree, Tree, l, r, b, t, dir)$
5. `Boolean:` QUERYRECT$(XYTree, l, r, b, t)$

It is possible to perform the entire sequence of the operations in $O(n^2 \log n)$ time, or in amortized $O(n \log n)$ time per operation.

Proof. Each of the operations on two dimensional segment trees listed in this chapter takes at most $O(n \log n + D)$ per operation, where D is the number of subtreedata bits cleared by the operation. If we begin with an empty two dimensional segment tree, then the total number of subtreedata bits cleared in all operations is at most as large as the number of bits which are set. The total number of subtreedata bits which are set is bounded by the running time of all operations not including operations involving the clearing of subtreedata bits. This running time is at most $O(n \log n)$ per operation, since the D in each of the running times comes exclusively from the clearing of subtreedata bits. Thus, the sum of all D values in all operations is at most $O(n \log n)$ times the number of operations, or $\sum D = O(n^2 \log n)$. So, amortized, each operation takes no more than $O(n \log n)$ time. □

Finally, it seems very likely that the need for amortized analysis can be removed, and that these operations all can be implemented with actual running times of $O(n \log n)$ or better. This improvement is discussed in Section 7.2.1.

5.5 Space Requirements

The space requirements on this kind of two dimensional segment tree are $O(n^2)$ after n operations, since there are $O(n)$ second level trees, each a one dimensional segment tree of size $O(n)$. This is in contrast to the space requirements of $O(n \log^2 n)$ on traditional segment trees [134].

5.6 Storing Interval Names

Traditional segment trees could store, not only the bounds of intervals, but also the names associated which each individual interval which had been inserted. A query function would then return a list of all intervals which a query point lies within.

In this section we define a variation on our kind of segment tree which recovers some of this original functionality. The purpose of this is to define a new QueryRange function which returns the name of any one of the intervals which has been stored within the given range, or returns NULL if there is no data stored there.

5.6.1 One Dimensional Segment Trees

A node of this new kind of segment tree is defined as follows:

```
SegTreeNode' =
   SegTreeNode':  left, right, parent
   ID: intname, subtreeint
   Num:  l, r
```

Here $intname$ is the name of any one inserted interval for which this is a canonical node, and $subtreeint$ is the name of any interval stored in the entire subtree rooted at this node. The value of $subtreeint$ is maintained as in Algorithm 27.

Algorithm 27 MAINTAINSUBTREEDATA$'(T)$

Given: A tree node, T, from an identifier storing one dimensional segment tree

```
1: T.subtreeint ← T.intname
2: if ( T is leaf ) then
3:    return
4: if ( T.subtreeint = NULL ) then
5:    T.subtreeint ← T.left.subtreeint
6: if ( T.subtreeint = NULL ) then
7:    T.subtreeint ← T.right.subtreeint
```

We define an analogous set of functions on this kind of tree:

- INSERTRANGE$'(T, l, r, intervalName)$

  Inserts the range $[\mu_l, \mu_r]$ with the name $intervalName$ into the segment tree. This is

implemented similarly to INSERTRANGE, the main difference being that the JOINDATA statement is removed. Note that nowhere in the preceding analysis is the JOINDATA statement used, so its removal does not affect running times.

- CLEARRANGE$'(T, l, r)$

  Clears from T the set of all points in the range $[\mu_l, \mu_r]$. This is implemented nearly identically to CLEARRANGE.

- ID: QUERYRANGE$'(T, l, r)$

  Queries T, returning the name of any interval stored within the range $[\mu_l, \mu_r]$. If no interval is stored here return NULL.

- UNIONTREES$'(T, S)$

  Store the union all the intervals in the two trees S and T together, overwriting T with the result. If two intervals names are unioned onto the same segment tree node of the destination, then either interval name may be stored there.

The functions INSERTRANGE$'$ and QUERYRANGE$'$ are defined as Algorithms 28 and 29.

Note that the JOINDATA statement has been removed from this variation on the original function. This is necessary, since the two nodes being joined may refer to two different intervals (previously the nodes simply referred to boolean variables).

Runtime Analysis

Now we analyze the running time of the new set of operations defined above on one dimensional segment trees. With the exception of the removal of the JOINDATA statement from the INSERTRANGE$'$ function, the control flow of each function above is identical to its previously defined counterpart.

Algorithm 28 INSERTRANGE$'(T, l, r, intervalName)$

Given: A tree node, T, from an interval storing one dimensional segment tree, a range $[\mu_l, \mu_r]$, and the name of the interval, $intervalName$, to be stored.

```
1: if ( (T = NULL) or (NOTWITHINRANGE(T, l, r)) ) then
2:    return
3: if ( ISCANONICAL(T, l, r)) ) then
4:    T.intname = intervalName
5: if ( ISCANONICALDEC(T, l, r)) ) then
6:    T.intname ← NULL
7:    if ( T.subtreeint = NULL ) then
8:       return
9: INSERTRANGE'(T.left, l, r, intervalName)
10: INSERTRANGE'(T.right, l, r, intervalName)
11: MAINTAINSUBTREEDATA'(T)
```

Algorithm 29 QUERYRANGE$'(T, l, r)$

Given: A tree node, T, from an interval storing one dimensional segment tree and a range $[\mu_l, \mu_r]$

Return: The name of any interval in segment tree T overlapping the range $[\mu_l, \mu_r]$, or NULL if there is none.

```
1: if ( (T = NULL) or (NOTWITHINRANGE(T, l, r)) ) then
2:    return NULL
3: if ( T.subtreeint = NULL ) then
4:    return NULL
5: if ( T.intname ≠ NULL ) then
6:    return T.intname
7: if ( ISCANONICAL(T, l, r) ) then
8:    return T.subtreeint
9: q ← QUERYRANGE'(T.left, l, r)
10: if ( q ≠ NULL ) then
11:    return q
12: return QUERYRANGE'(T.right, l, r)
```

The removal of JOINDATA means that the data upon which it would have operated remains in the tree until specifically deleted by the CLEARRANGE′ function. This changes the value of D from the analysis of the running time of CLEARRANGE but does not change its amortized running time, since the deletion of the data was merely delayed.

The running time of the QUERYRANGE′ function is unaffected by the removal of JOINDATA since it can look at the value of $subtreeint$ higher up in the tree.

Thus it follows that the amortized running times of these functions are identical to their previously defined counterparts.

5.6.2 Two Dimensional Segment Trees

Here we will define the following operations on two dimensional segment trees. In most cases these functions are implemented nearly indentically to their previously implemented counterparts.

- INSERTRECT′$(T, l, r, b, t, rectName)$

  Insert the rectangle $[\mu_l, \mu_r] \times [\mu_b, \mu_t]$ into the two dimensional segment tree T, storing it with the name $rectName$.

- INSERTSTRIPEDRECT′$(XYTree, Tree, l, r, b, t, sDir)$ Stripe the rectangle $[\mu_l, \mu_r] \times [\mu_b, \mu_t]$ in the two dimensional segment tree $XYTree$ according to the contents of the one dimensional segment tree $Tree$. The direction of striping is either x or y as indicated by $sDir$.

- CLEARRECT′(T, l, r, b, t)

  Clear all points from the two dimensional segment tree T which lie within the rectangle $[\mu_l, \mu_r] \times [\mu_b, \mu_t]$.

- ID: QUERYRECT′(T, l, r, b, t)

  Query the two dimensional segment tree T, returning the name of any rectangle overlapping $[\mu_l, \mu_r] \times [\mu_b, \mu_t]$.

- PROJECTRECT′$(XYTree,\ Tree, l, r, b, t, pDir)$

  Project the contents of the rectangle $[\mu_l, \mu_r] \times [\mu_b, \mu_t]$ within the two dimensional segment tree $XYTree$ onto one of the axes, storing the result in the one dimensional segment tree $Tree$. The choice of axis, either x or y, is indicated by $sDir$.

We define INSERTRECT′ and QUERYRECT′ in Algorithm 30 and 31.

Algorithm 30 INSERTRECT′$(XYTree, l, r, b, t, rectName)$

Given: A tree node, $XYTree$, from a rectangle storing two dimensional segment tree and a rectangle $[\mu_l, \mu_r] \times [\mu_b, \mu_t]$

```
1: if ( (XYTree = NULL) or (NOTWITHINRANGE(XYTree, b, t)) ) then
2:     return
3: if ( ISCANONICALDEC(XYTree, b, t) ) then
4:     return
5: if ( ISCANONICALANC(XYTree, b, t) ) then
6:     INSERTRANGE′(XYTree.projtree, b, t, rectName)
7: if ( ISCANONICAL(XYTree, b, t) ) then
8:     INSERTRANGE′(XYTree.perptree, b, t, rectName)
9:     INSERTRANGE′(XYTree.projtree, b, t, rectName)
10: INSERTRECT′(XYTree.left, l, r, b, t, rectName)
11: INSERTRECT′(XYTree.right, l, r, b, t, rectName)
```

Note we assume that the names of rectangles and the names of intervals are both identifier datatypes which can be assigned interchangeably.

Runtime Analysis

The structure of the operations defined above on our new variation on two dimensional segment trees in all cases is identical to that of their previously defined counterparts.

Algorithm 31 QUERYRECT$'(XYTree, l, r, b, t)$

Given: A tree node, $XYTree$, from a rectangle storing two dimensional segment tree and a rectangle $[\mu_l, \mu_r] \times [\mu_b, \mu_t]$

Return: The name of any rectangle in segment tree $XYTree$ overlapping the rectangle $[\mu_l, \mu_r] \times [\mu_b, \mu_t]$, or NULL if there is none.

1: **if** (($XYTree =$ NULL) or (NOTWITHINRANGE$(XYTree, b, t)$))) **then**
2: **return** NULL
3: **if** (ISCANONICAL$(XYTree, b, t)$) **then**
4: **return** QUERYRANGE$'(XYTree.projtree, l, r)$
5: $q \leftarrow$ QUERYRANGE$'(XYTree.perptree, l, r)$
6: **if** ($q \neq$ NULL) **then**
7: **return** q
8: $q =$ QUERYRECT$'(XYTree.left, l, r, b, t)$
9: **if** ($q \neq$ NULL) **then**
10: **return** q
11: **return** QUERYRECT$'(XYTree.right, l, r, b, t)$

These depend on the running time of the operations on one dimensional segment trees. The amortized running times of the operations on one dimensional segment trees is unchanged and so the amortized running times of the operations on two dimensional segment trees is likewise unchanged.

Chapter 6

Experimental Results

As part of this thesis, the algorithm described in Chapter 4 was implemented in C. This included an implementation of the data structure described in Chapter 5, and a randomized binary space partition implementation. This implementation served several purposes.

First and foremost, the implementation demonstrated that the algorithm is fairly easy to implement. Everything described in this chapter required less than 8000 lines of source code, including all header files, user interface files, files generating and running all test cases, and files implementing the binary space partition. This was done without the aid of any libraries or object-oriented constructs. The minimum link path algorithm itself, including the segment tree data structure, the block and face data structures, the priority queue, the sweep plane, and their associated functions, required less than 4300 lines of C. Table 6.1 gives a count of the lines in these individual files.

Additionally, a number of randomly generated test cases, with up to 3900 faces, were run with this implementation. This was intended to demonstrate the experimental running time, and to compare this to the theoretical bound of $O(\beta n \log n)$. The results of this comparison appear in Section 6.2.

Lines	Words	Characters	Filename
320	925	6426	datatypes.h
310	822	7529	block.c
41	170	1366	block.h
104	268	2295	bounds.c
16	80	620	bounds.h
595	1888	17025	generate.c
50	203	1476	generate.h
166	409	3366	queue.c
25	73	546	queue.h
897	2834	21148	segtree.c
93	443	3439	segtree.h
466	1420	12732	space.c
36	123	1157	space.h
1014	3315	26601	subface.c
77	374	3339	subface.h
61	130	1363	sweepplane.c
9	37	322	sweepplane.h
4280	13514	110750	total

Table 6.1: A count of the number of lines, words, and characters in each of the implementation files

6.1 Setup

The algorithm described in Chapter 4 was implemented in C. The source code of this implementation appears in the Appendix. This was compiled under the GNU C compiler, gcc version 3.4.4 20050721 (Red Hat 3.4.4-2).

This experiment was run on a dedicated node within a beowulf cluster in the Computer Science Department at Dartmouth College (known as the Jefferson cluster). The node has a pair of 2.8 GHz Intel XEON processor, provides 6 Gigabyes of RAM, and is running RedHad Linux 9.

All tests performed here are in a space of size $1000 \times 1000 \times 1000$. All obstacles and starting and finishing points have integer coordinates.

6.1.1 Initial Tests

Initial tests involved some very simple random experiments in three dimensional space. These were generated by first randomly placing start and finish points, and then choosing pairs of coordinate to serve as the corners of rectangular obstacles. If a new obstacle intersected with a previously inserted object, then it was discarded.

As perhaps might be expected, a great majority of these initial test cases had trivial solutions, with just two bends. The experiment then went through a series of modifications in an attempt to be able to randomly generate large test cases with non-trivial solutions.

6.1.2 Inserting Obstacles

In an attempt to increase the minimum bend distance between the start and finish points, the two points were first moved to opposite corners of space. Second, the obstacles were added in two phases.

In the first phase, randomly chosen pairs of points were selected on a wall of space. This pair of points was used to define a rectangular obstacle of width one lying against the wall. Again, if a new obstacle intersected with a previously inserted object, then the new obstacle was discarded.

In the second phase, a random face was selected from the list of previously inserted faces. The face was then perpendicularly extended a random distance between one and distance to the opposite wall. If this caused the obstacle to intersect with another obstacle, then this distance was repeatedly halved, until there was no conflict. The result was to effectively "grow" a rectangular obstacle out of an existing obstacle.

In this way, a series of obstacles were first placed flat against the walls of space, and then another series of obstacles were grown out of existing obstacles, until the space was filled with an intricate maze of obstacles separated by narrow passageways.

Unfortunately, this procedure failed to do much to increase the minimum link-distance between the start and finish points. The majority of tests run under this model had solutions with between two and four bends. Upon closer examination, the reason behind this became apparent. In most cases a minimum link path could run along the majority, or the entirety of some or all of the edges in space.

In order to remove the simplest paths from these test cases, another modification was made to the experimental setup. Prior to the insertion of any other obstacles, a series of twelve unit sized ($1 \times 1 \times 1$) obstacles was added. One of these was placed at the center of each edge of space, thereby eliminating the possibility that a path could run the entire length of an edge of space. This modification immediately increased the number of bends in any minimum link path to at least four.

The resulting test cases typically have minimum link paths with between four and six bends.

6.1.3 Conflict Resolution

At this point it should be mentioned that a number of issues concerning the correctness of the obstacles are being handled. As described in Section 4.1.1, the obstacles which are given to the binary space partition algorithm are represented as a set of faces. They are not stored as three dimensional blocks. New obstacle faces which are added to this set of faces must not only avoid intersecting existing faces, but must also avoid being located inside existing obstacles or enclosing existing obstacles (this would confound the notion of which blocks are inside and outside of the obstacles).

The method of "growing" obstacles out of existing faces avoids the possibility that these new faces will be inside of existing obstacles, with one exception. It is still possible that a new obstacle along the wall will be inside an existing obstacle, or will enclose an existing obstacle. By associating a segment tree with each wall, and by using the INSERTRECT and QUERYRECT functions, however, this possibility is avoided.

Another requirement of the minimum link path algorithm is that pairs of obstacle faces should not be adjacent to and partially overlap with each other, regardless of whether they are facing in the same direction or not. The main reason for this restriction is that such overlapping faces are not allowed by the obstacle model (See Section 4.1.1). New adjacent overlapping faces which face each other might perhaps be split into quadrants to permit their insertion, but for simplicity this is not done.

Thus, all new faces are tested for intersection with and for partial overlap with other faces, and any such new faces are disallowed, and a conflict is reported.

One final possibility is that a pair obstacle faces might be adjacently facing each other and completely overlap so that all boundaries are flush (that is, they have all the same coordinates). In this case, then they can both be removed, since this eliminates the overlap and does not create any new conflicts.

So any new face which adjacently faces and completely overlaps with an existing face is removed, along with the adjacent face and no conflict is reported. Note that this always happens during a "grow" operation.

6.1.4 Forcing a larger β

When timings were run on some small test cases, it became apparent that the value of β varied almost linearly with the value of n. See Figure 6.1 for this comparison.

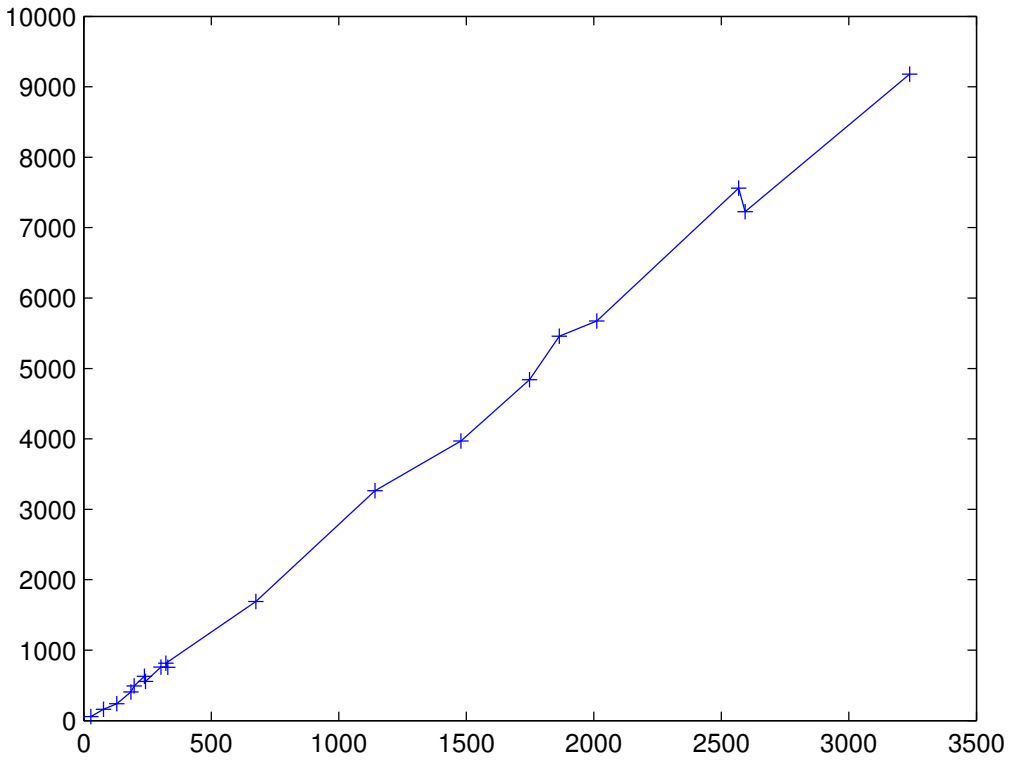


Figure 6.1: The number of faces plotted against the number of nodes in a BSP tree. This is done without the β-increasing heuristics described in Section 6.1.4. Initial test cases show the relationship is close to linear.

In order to vary the value of β, modifications to each stage of the obstacle insertion algorithm were made.

First, when new flat obstacles are randomly attached to the walls of space, they are given a "shrink factor". This value is the number of times the size of the obstacle will shrink to $3/4$ of its previous size (in both directions). Thus, by varying this amount, it is possible to randomly attach larger or smaller obstacles to the walls.

Second, when new obstacles are randomly grown out of existing faces, they are given an "extend factor". This value is the number of times the gap between the edge of the randomly chosen obstacle extension and the opposite wall is decreased to $3/4$ of its previous distance. By varying this amount it is possible to have longer or shorter extensions.

The idea behind these modifications is that larger shrink and larger extend factors will together result in longer skinnier obstacles. In this way, the size of the Binary Space Partition should increase with the same number of faces.

Figure 6.2 shows that tests under this model, with up to 3500 faces, were moderately successful in increasing the value of β, achieving about $\beta = \Theta(n \log n)$.

6.2 Experimental Results

Figure 6.2 plots the number of faces n against the number of nodes in the BSP tree, β. These experiments were run while the β increasing heuristics, described in Section 6.1.4 were in place.

Figure 6.3 demonstrates the running time in terms of n, the number of faces, and Figure 6.4 demonstrates the running time in terms of β the size of the BSP tree. In both cases the β increasing heuristic was in place. The experiments appear to demonstrate running times of $O(n \log^2 n)$ and $O(\beta \log \beta)$ respectively.

According to these results, the implementation appears to run substantially faster than the predicted $O(\beta n \log n)$ bound. There may be a number of reasons for this.

One possibility is small bend distance of the solutions that were found (In nearly all

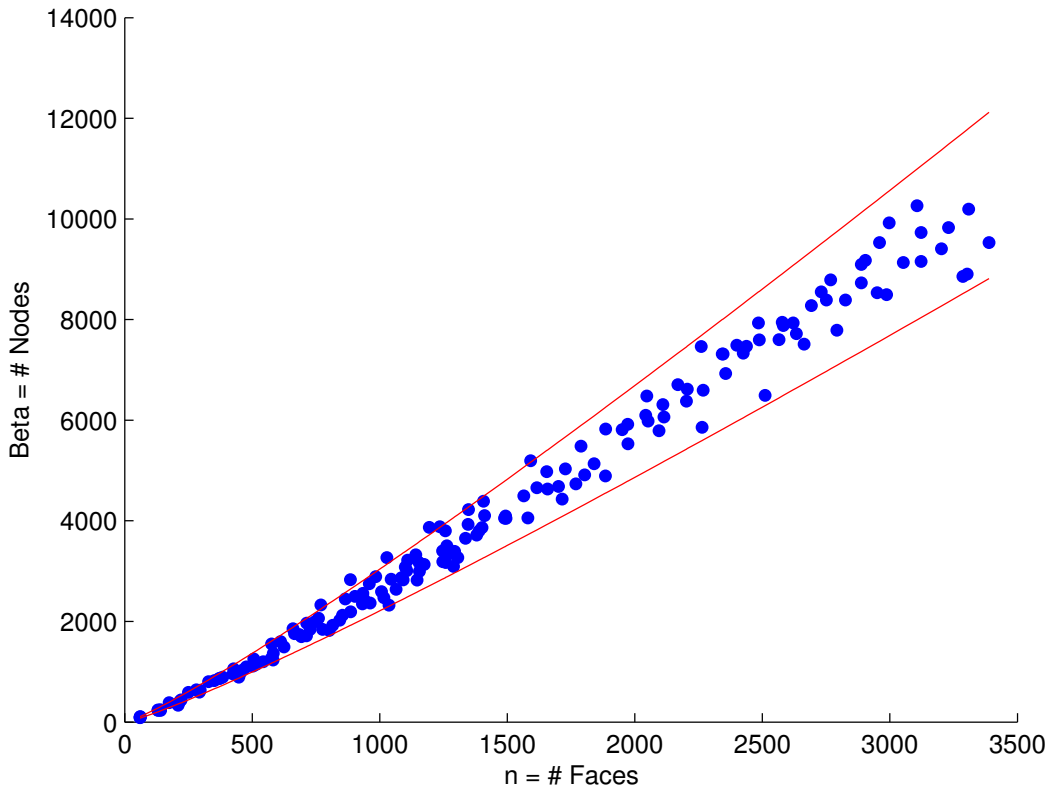


Figure 6.2: This figure demonstrates an attempt to increase β. The number of faces is plotted against the number of nodes in a BSP tree, while using the β-increasing heuristics described in Section 6.1.4. Although the theoretic bound of β is $O(n^{3/2})$, this method experimentally appears to have achieved $\beta = \Theta(n \log n)$. The two solid lines are plots of $\Theta(n \log n)$ with different constants.

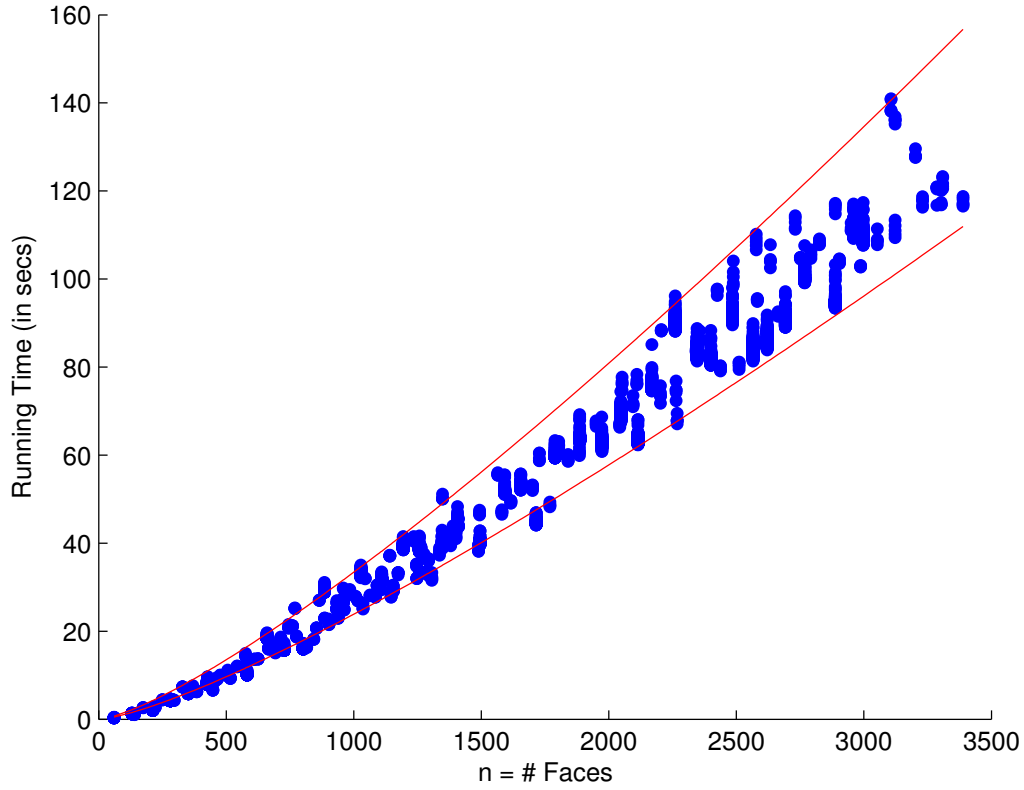


Figure 6.3: The number of faces plotted against the running time with the β-increasing heuristic. Although the theoretic bound of the running time is $O(n^{5/2} \log n)$, this method experimentally appears to have achieved a running time of $\Theta(n \log^2 n)$. The two solid lines are plots of $\Theta(n \log^2 n)$ with different constants.

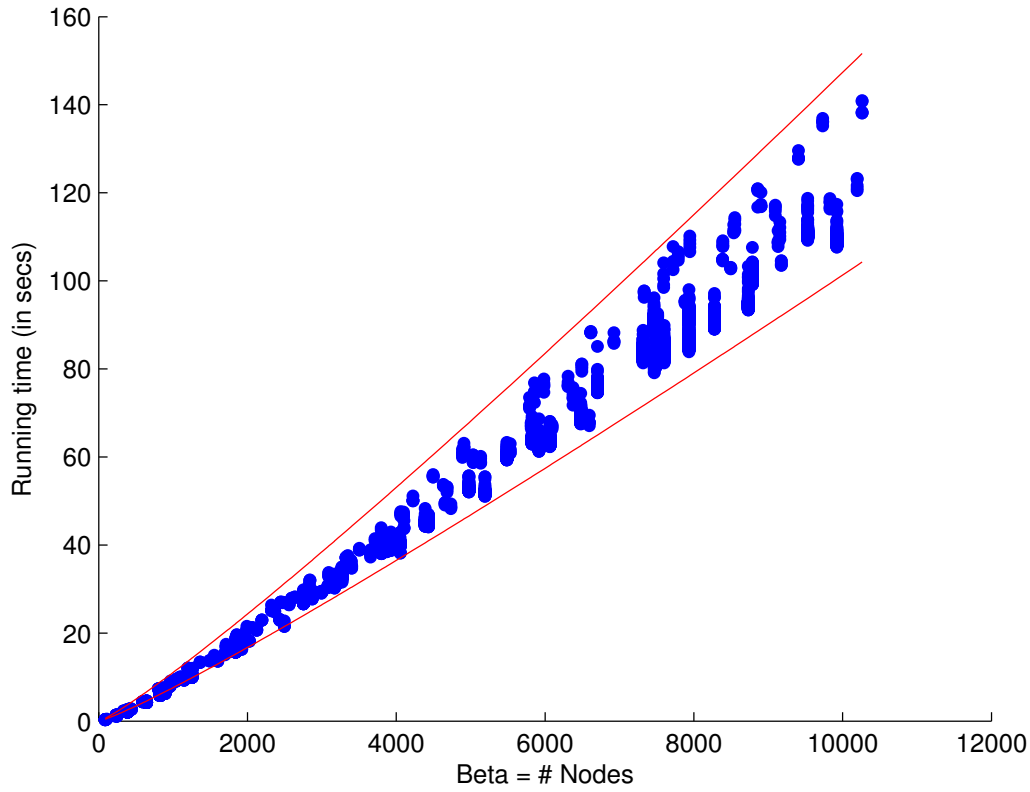


Figure 6.4: The number of nodes, β, plotted against the running time. Although the theoretic bound on the running time is $O(\beta n \log \beta)$, This method experimentally appears to have achieved a running time of $\Theta(\beta \log \beta)$. The two solid lines are plots of $\Theta(\beta \log \beta)$ with different constants.

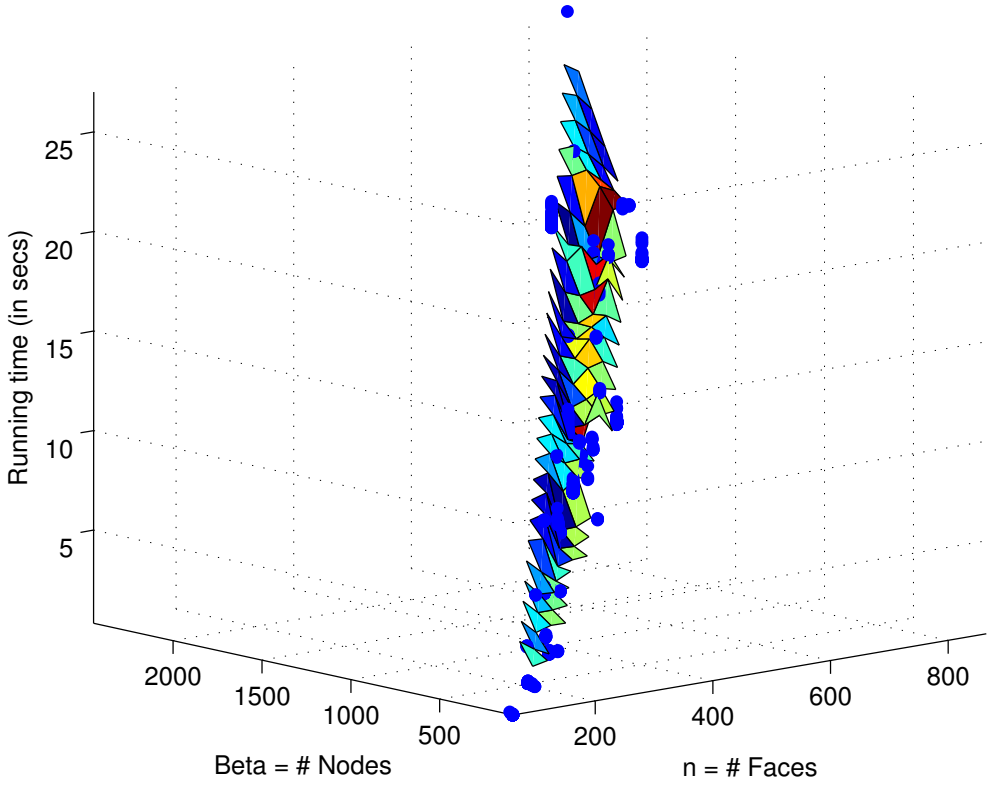


Figure 6.5: The running times of up to 800 faces with different values of β, with the β-increasing heuristic. This is done while using the β increasing heuristics described in Section 6.1.4. It can be seen in this example that both β and n contribute to the running time.

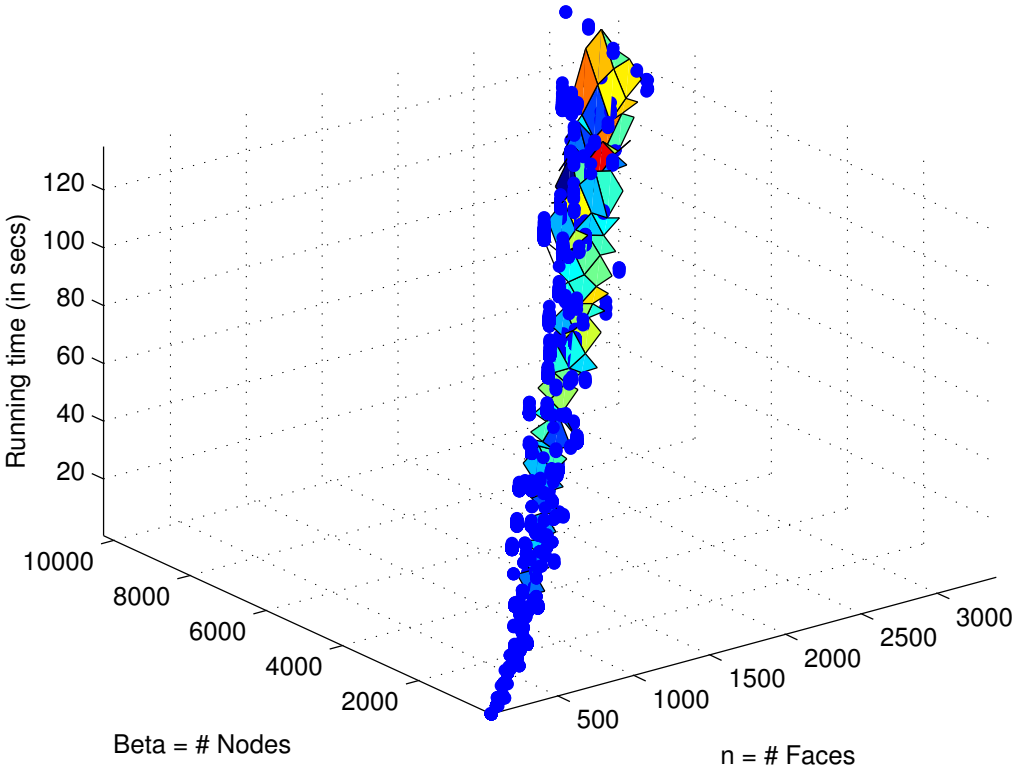


Figure 6.6: The running times of up to 3000 faces with different values of β, with the β-increasing heuristic. In this example, it is difficult to see if β contributes to the running time, independently of n.

cases the largest bend distance was six). This suggests that further analysis, taking into account the link-distance of the minimum link path, might yield a tighter bound on the running time.

It is also possible that the low complexity of the test cases generated could be a factor. A more sophisticated obstacle generating method would be needed to verify this.

Figure 6.5 shows that both n and β contribute to the running time when there are up to 800 faces. This distinction becomes harder to see when there are more faces. Figure 6.6 shows this lost distinction when there are up to 3000 faces.

Chapter 7

Conclusions and Future Work

In this dissertation several algorithms solving path planning problems under the link-distance measure have been discussed. Those results are reiterated in this chapter.

7.1 Contributions

In this thesis, we have discussed the Minimum Bends Traveling Salesman Problem, where the objective is to visit a number of points in the plane using a tour containing the minimum number of straight line segments. We have given a logarithmic approximation algorithm solving this problem, and we have proven that this general problem is NP-Complete, by reducing Set Cover to this problem. We further have given a 2-approximation algorithm for this problem when all paths are restricted to being rectilinear (axis-parallel). Finally, when the paths are restricted to being rectilinear and there are no two points lined up vertically or horizontally, we give an approximation algorithm that comes within two bends of the optimal solution.

We have also discussed the Minimum Link-Distance problem, where the objective is to find a path among obstacles between two points consisting of the minimum number

of straight line segments. We have given an algorithm that solves this problem in three dimensions, when paths are restricted to being rectilinear, and obstacles are also rectilinear. This algorithm relies on the existence of a data structure supporting the efficient insertion and removal of all points within a rectangle in the plane.

We give a data structure that supports the operations needed by the previously mentioned algorithm. This data structure is a variation on the segment tree. The operations include the insertion and removal of all points within a rectangle, the projection of all points within a rectangle onto an axis, and the insertion of a set of rectangles that share coordinates in one direction, but have different coordinates in another direction, as specified by the set of intervals in a one dimensional segment tree.

Finally, an implementation of the Minimum Link-Distance algorithm, appearing in the appendix, demonstrates that it is not terribly difficult to implement. Furthermore, experiments running this implementation have shown empirically that the algorithm may actually run faster than what has been proven here. Specifically, if the link-distance of the solution is taken into consideration, this may yield an improvement in theoretical bound.

7.2 Future work

The solutions to the problems we have discussed in this dissertation still leave open many areas for further progress. Perhaps the most immediate area for further study exists in extending the Minimum Link-Distance Problem into higher dimensions. Many techniques used here are readily applicable in higher dimensions, and are discussed below in Section 7.2.2 and Section 7.2.3. Also it appears that a small modification will remove the need for amortized analysis as discussed in Section 7.2.1

7.2.1 Removing Amortized Analysis and Running Time Improvements

It seem very likely that the use of amortized analysis in INSERTRANGE and CLEARRANGE and consequently in the two dimensional operations can be removed with a minor change. Recall that the need for amortized analysis results entirely from the possibility that $\Omega(n)$ different $subtreedata$ bits might need to be cleared in a single call to either of the functions CLEARRANGE or INSERTRANGE. In each case this is because the entire subtree rooted at a node might need to be cleared.

This need can apparently be avoided if a $subtreedata$ bit may be cleared in a node, thereby indicating that the entire subtree rooted at that node has been cleared out, without operating on any of its descendants. Under this model, it is not necessary to clear all $subtreedata$ bits in an entire subtree when a CLEARRANGE or INSERTRANGE operation is performed. Instead the $subtreedata$ bit is cleared and future operations that access the subtree will examine that setting, and take it into account.

Another way to think about this is that every node is in one of three states. Its range is entirely cleared (its $subtreedata$ bit is cleared), its range is entirely filled (its $data$ bit is set), or its range is partially filled in ($subtreedata$ is set, but $data$ is cleared). In the first two cases, any data in the subtree can be ignored by a query. Only in the third case is it necessary for a query to examine the subtree. Operations that modify the tree only need to assign this state to a bounded number of nodes, and thus the children of such nodes can all be assigned to one of the first two states.

The author is grateful to Jon Bentley and to Günter Rote for suggesting this improvement [19, 142].

A thorough analysis of this improvement is left for future work.

7.2.2 Higher Dimensional Data Structures

The methods described in this chapter extend naturally into higher dimensions. Inserting or clearing all data in a d-dimensional hyperrectangle is a straightforward extension of the INSERTRECT and CLEARRECT operations on d-dimensional segment trees.

A d-dimensional projection operation involves removing one of the dimensions in a d-dimensional segment tree. A d-dimensional segment tree contains d layers, and thus a projection operation will remove one of these layers.

Similarly, a d-dimensional stripe or sweep operation involves adding a dimension in a d-dimensional segment tree. That is, a $d-1$-dimensional tree is swept through an existing d-dimensional tree, and must be unioned onto the appropriate nodes of that tree.

A more careful description and analysis of these operations is needed. This is left for future work.

7.2.3 Higher Dimensional Minimum Link Path

The development of higher dimensional data structures would consequently permit the extension of the minimum link path algorithm into higher dimensions. Most of the techniques described in Chapter 4 can be extended into higher dimensions in a straightforward manner.

It bears mention, however, that the number of projections and sweep/stripe calls grows exponentially with the number of dimensions. The notion of `Class A`, `Class B`, and `Class C` point sets used in Chapter 4 corresponds to projecting two dimensional data onto two, one, and zero dimensional space respectively, and then expanding that data again to fill a two dimensional rectangle.

One way to think of a projection is that it removes some number of dimensions from the data. In d dimensions it becomes necessary to perform 2^d projections by removing all possible subsets of the d dimensions, and then to expand each of these back into d

dimensions.

Another requirement in extending the algorithm into higher dimensions is the efficient computation of higher dimensional binary space partitions. Dumitrescu, Mitchell, and Sharir have recently discussed this [44].

7.2.4 Spanning Tree

A consideration of the spanning tree problem under this metric will reveal that it is a rich area which deserves further study. Even when restricted to rectilinear paths in two dimensions, there exist many variations on the problem, including considerations about how to count bends at T-intersections, and when lines are crossing. Solutions to some of these variations follow readily from the results listed here, while other variations appear to be quite difficult.

Additionally, the spanning tree problem would have direct applications in VLSI. Consider that on circuit board, it is often several points that must be connected together, and not always two points.

This appears to be a very worthwhile area for future study.

The author is grateful to Joseph S. B. Mitchell for many discussions illuminating the intricacy of this problem [121].

7.2.5 An Approximation Algorithm for the Min Link Path Problem

Some of our work has focused on finding a constant time approximation algorithm for the Minimum Link Path Problem, which yields a better running time than the one described here.

One part of this work has involved showing that an optimal path can be adjusted so that it touches an obstacle face at least once every other link. It then seems likely that one

can connect together the faces that are reachable within this number of bends, resulting in a graph over the obstacle faces. Using Dijkstra's algorithm to search for a shortest path through this graph [39], and paying a constant number of extra bends at each face, would then give an approximation of an optimal path.

It seems quite believable that these techniques can be used to develop an approximation algorithm that runs faster than the theoretical analysis of the algorithm described in Chapter 4. However, given that the experimental analysis appears to demonstrate a faster running time than the theoretical analysis, such an approximation algorithm may or may not have practical utility.

7.2.6 Lower bounds

It is interesting to consider the lower bounds on these problems. Proofs exist giving an $\Omega(n \log n)$ lower bound on the two dimensional link-distance problem [123, 95], but these results to not readily extend into higher dimensions. Improvements in lower bounds have typically been much more difficult to find than improvements in the running time of algorithms. Thus, this remains a daunting task.

7.2.7 Improved Analysis

Finally, it is intriguing to consider that a more careful analysis of the Minimum Link Path algorithm described in Chapter 4 may yield a tighter bound on the running time. One prospect to consider is including k, the number of bends in a solution, as a factor in the running time. Recall that several authors have given algorithms where the link-distance of the solution is a factor in the running time [14, 50, 123, 140].

A more careful analysis is left for future work.

7.2.8 Additional Test Cases

In addition to further analysis, the hypothesis that k, the number of bends in the solution, contributes to the overall running time might be tested by devising a series of tests forcing a higher k. These tests have not been performed here, however it would not be terribly difficult to do this.

Bibliography

[1] Foto N. Afrati, Stavros S. Cosmadakis, Christos H. Papadimitriou, George Papageorgiou, and Nadia Papakostantinou. The complexity of the travelling repairman problem. *Informatique Theoretique et Applications*, pages 79–87, 1986.

[2] Alok Aggarwal, Don Coppersmith, Sanjeev Khanna, Rajeev Motwani, and Baruch Schieber. The angular-metric traveling salesman problem. *SIAM Journal on Computing*, 29(3):697–711, June 2000.

[3] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows : Theory, Algorithms, and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[4] Muhammad H. Alsuwaiyel and D. T. Lee. Minimal link visibility paths inside a simple polygon. *Comput. Geom. Theory Appl.*, 3(1):1–25, 1993.

[5] Muhammad H. Alsuwaiyel and D. T. Lee. Finding an approximate minimum-link visibility path inside a simple polygon. *Inf. Process. Lett.*, 55(2):75–79, 1995.

[6] Esther M. Arkin, Michael A. Bender, Erik D. Demaine, Sándor P. Fekete, Joseph S. B. Mitchell, and Saurabh Sethia. Optimal covering tours with turn costs. In *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms*, pages 138–147, Washington, DC, 2001.

[7] Esther M. Arkin, Michael A. Bender, Erik D. Demaine, Sándor P. Fekete, Joseph S. B. Mitchell, and Saurabh Sethia. Optimal covering tours with turn costs. *SIAM Journal on Computing*, To appear. Also appearing online in *http://citebase.eprints.org/*.

[8] Esther M. Arkin, Yi-Jen Chiang, Joseph S. B. Mitchell, Steven S. Skiena, and Tae-Cheon Yang. On the maximum scatter traveling salesman problem. *SIAM Journal on Computing*, 29(2):515–544, 1999.

[9] Esther M. Arkin, Sándor P Fekete, and Joseph S. B. Mitchell. Approximation algorithms for lawn mowing and milling. *Computational Geometry: Theory and Applications*, 17(1-2):25–50, October 2000.

[10] Esther M. Arkin and Refael Hassin. Approximation algorithms for the geometric covering salesman problem. *Discrete Applied Mathematics*, 55:197–218, 1994.

[11] Esther M. Arkin, Joseph S. B. Mitchell, and Christine D. Piatko. Minimum-link watchman tours. *Information Processing Letters*, 86(4):203–207, 2003.

[12] Esther M. Arkin, Joseph S. B. Mitchell, and Subhash Suri. Optimal link path queries in a simple polygon. In *Proceedings of the 3rd ACM-SIAM Symposium on Discrete Algorithms*, pages 269–279, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.

[13] Sanjeev Arora. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *JACM: Journal of the ACM*, 45, 1998.

[14] Tetsuo Asano. An efficient algorithm for finding the region reachable within k bends. *Transactions of the IECE of Japan*, E-68(12):831–835, December 1985.

[15] Alexander I. Barvinok, Sándor P. Fekete, David S. Johnson, Arie Tamir, Gerhard J. Woeginger, and Russell Woodroofe. The geometric maximum traveling salesman problem. *Journal of the ACM*, 50(5):641–664, 2003.

[16] Alexander I. Barvinok, David S. Johnson, Gerhard J. Woeginger, and Russell Woodroofe. The maximum traveling salesman problem under polyhedral norms. In *In Sixth International Conference on Integer Programming and Combinatorial Optimization*, pages 195–201. Springer-Verlag, LNCS 1412, 1998.

[17] B. G. Baumgart. A polyhedral representation for computer vision. *Proc. AFIPS Natl. Comput. Conf.*, 44:589–596, 1975.

[18] Jon L. Bentley. Solutions to Klee's rectangle problems. Technical report, Carnegie-Mellon University, 1977.

[19] Jon L. Bentley, 2005. Personal Communication.

[20] Jon L. Bentley and Derick Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Transactions on Computers*, C-29(7):571–577, July 1980.

[21] Mark de Berg. On rectilinear link distance. *Computational Geometry: Theory and Applications*, 1(1):13–34, 1991.

[22] Mark de Berg, Marc J. van Kreveld, Bengt J. Nilsson, and Mark H. Overmars. Shortest path queries in rectilinear worlds. *International Journal of Computational Geometry and Applications*, 2(3):287–309, 1992.

[23] Mark de Berg, Marc J. van Kreveld, Mark H. Overmars, and Otfried Schwarzkopf. *Computational Geometry, Algorithms and Applications*, chapter 10. Springer, 2000.

[24] Avrim Blum, Prasad Chalasani, Don Coppersmith, Bill Pulleyblank, Prabhakar Raghavan, and Madhu Sudan. The minimum latency problem. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pages 163–172, May 1994.

[25] Prosenjit Bose and Godfried Toussaint. Computing the constrainted euclidean geodesic and link centre of a simple polygon with applications. In *Proceedings of Computer Graphics International 1996*, pages 102–111. IEEE, 1996.

[26] Hervé Brönnimann and Michael T. Goodrich. Almost optimal set covers in finite VC-dimension. *Discrete and Computational Geometry*, 14:263–279, 1995.

[27] Rainer. E. Burkard, Vladimir G. Deĭneko, René van Dal, Jack A. A. van der Veen, and Gerhard J. Woeginger. Well-solvable special cases of the traveling salesman problem: A survey. *SIAM Review*, 40:496–546, September 1998.

[28] Rainer E. Burkard and W. Sandholzer. Efficiently solvable special cases of bottleneck traveling salesman problems. *Discrete Applied Mathematics*, 32:61–76, 1991.

[29] Vijay Chandru, Subir Kumar Ghosh, Anil Maheshwari, V. T. Rajan, and Sanjeev Saluja. NC-algorithms for minimum link path and related problems. *Journal of Algorithms*, 19:173–203, 1995.

[30] Bernard Chazelle. *Computational Geometry and Convexity*. PhD thesis, Yale University, New Haven, CT, 1980. Report CMU-CS-80-150, Computer Science Department, Carnegie Mellon.

[31] Bernard Chazelle. Triangulating a simple polygon in linear time. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 220–229, 1990.

[32] Bernard Chazelle and David P. Dobkin. *Computational Geometry*, pages 63–133. North-Holland, Amsterdam, 1985.

[33] Bernard Chazelle, Herbert Edelsbrunner, Leonidas J. Guibas, and Micha Sharir. Algorithms for bichromatic line segment problems and polyhedral terrains. *Algorithmica*, 11:116–132, 1994.

[34] Nicos Christofides. Worst case analysis of a new heuristic for the traveling salesman problem. Technical Report Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA, 1976.

[35] Vašek Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, August 1979.

[36] Michael J. Collins. Covering a set of points with a minmum number of turns. *International Journal of Computational Geometry and Applications*, 14(1/2):105–114, 2004.

[37] Michael J. Collins and Bernard M. E. Moret. Improved lower bouns for the link length of rectilinear spanning paths in grids. *Information Processing Letters*, 68:317–319, 1998.

[38] Gautam Das and Giri Narasimhan. Geometric searching and link distance. In *Proceedings of the Second Workshop on Algorthms and Data Structures*, volume 591 of *LNCS*, pages 261–272. Springer Verlag, LNCS 591, 1991.

[39] E. W. Dijkstra. "A note on two problems in connection with graphs". *Numerische Mathematik*, 1:260–271, 1959.

[40] Hristo N. Djidjev, Andrzej Lingas, and Jörg-Rüdiger Sack. An $O(n \log n)$ algorithm for computing the link center of a simple polygon. *Discrete and Computational Geometry*, 8:131–152, 1992.

[41] Robert L. (Scot) Drysdale III, Clifford Stein, and David P. Wagner. An $O(n^{5/2} \log n)$ algorithm for the rectilinear minimum link-distance problem in three dimensions. In *Proceedings of the 17th Canadian Conference on Computational Geometry*, pages 94–97, 2005.

[42] Rong-Chii Duh and Martin Fürer. Approximation of k-set cover by semi-local optimization. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 256–264, 1997.

[43] Adrian Dumitrescu and Joseph S. B. Mitchell. Approximation algorithms for TSP with neighborhoods in the plane. *Journal of Algorithms*, 48:135–159, 2003.

[44] Adrian Dumitrescu, Joseph S. B. Mitchell, and Micha Sharir. Binary space partitions for axis-parallel segments, rectangles, and hyperrectangles. *Discrete and Computational Geometry*, 31(2):207–227, February 2004.

[45] Herbert Edelsbrunner. Dynamic data structures for orthogonal intersection queries. Technical Report F59, Tech. Univ. Graz, Austria, 1980.

[46] Herbert Edelsbrunner. A new approach to rectangle intersections - part I. *International Journal of Computer Mathematics*, 13:209–219, 1983.

[47] Herbert Edelsbrunner. A new approach to rectangle intersections - part II. *International Journal of Computer Mathematics*, 13:221–229, 1983.

[48] Herbert Edelsbrunner and Hermann A. Maurer. On the intersection of orthogonal objects. *Inform. Process. Lett.*, 13:177–181, 1981.

[49] Jenö Egerváry. Matrixok kombinatorius tulajdonságairól. *Matematikai és Fizikai Lápok*, 38:19–25, 1931. (in Hungarian).

[50] Hossam Ahmed ElGindy. *Hierarchical decomposition of polygons with applications*. PhD thesis, McGill University, 1985.

[51] Uriel Feige. A threshhold of $\ln n$ for approximating set cover. *Journal of the ACM*, 45(4):634–652, July 1998.

[52] Sándor P. Fekete. Simplicity and hardness of the maximum traveling salesman problem under geometric distances. In *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms*, Baltimore, MD, 1999.

[53] M. L. Fisher, George L. Nemhauser, and Laurence A. Wolsey. An analysis of approximations for finding a maximum weight hamiltonian circuit. *Operations Research*, 27(4):799–809, July–August 1979.

[54] Robert Fitch, Zack Butler, and Daniela Rus. 3D rectilinear motion planning with minimum bend paths. In *International Conference on Intelligent Robots and Systems*, Wailea, Maui, HI, 2001.

[55] Michael L. Fredman and Bruce Weide. On the complexity of computing the measure of $\cup_1^n [a_i, b_i]$. *Communications of the ACM*, 21(7):540–544, July 1978.

[56] Michael R. Garey, Ronald L. Graham, and David S. Johnson. Some NP-complete geometric problems. In *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*, pages 10–22, 1976.

[57] Robert S. Garfinkel. Minimizing wallpaper waste, part I: a class of traveling salesman problems. *Operations Research*, 25:741–751, 1977.

[58] Robert S. Garfinkel and K. C. Gilbert. The bottleneck traveling salesman problem: Algorithms and probabilistic analysis. *Journal of the ACM*, 25(3):435–448, July 1978.

[59] Paul C. Gilmore and Ralph E. Gomory. Sequencing a one state-variable machine: a solvable case of the traveling salesman problem. *Operations Research*, 12:655–679, 1964.

[60] Olivier Goldschmidt, Dorit S. Hochbaum, and Gang Yu. A modified greedy heuristic for the set covering problem with improved worst case bound. *Information Processing Letters*, 48:305–310, 1993.

[61] Daniel H. Greene. The decomposition of polygons into convex parts. In Franco P. Preparata, editor, *Advances in Computing Research*, volume 1, pages 235–259. JAI Press Inc., 1983.

[62] Joachim Gudmundsson and Christos Levcopoulos. A fast approximation algorithm for TSP with neighborhoods. *Nordic Journal of Computing*, 6:469–488, 1999.

[63] Joachim Gudmundsson and Christos Levcopoulos. Hardness result for TSP with neighborhoods. Technical Report LU-CS-TR:2000-216, Department of Computer Science, Lund University, Sweden, 2000.

[64] Leonidas J. Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. *ACM Trans. Graph.*, 4:74–123, 1985.

[65] Gregory Gutin and Abraham P. Punnen. *The Traveling Salesman Problem and Its Variations*, volume 12 of *Combinatorial Optimization*. Kluwer Academic Publishers, 2002.

[66] Rafael Hassin and Shlomi Rubinstein. An approximation algorithm for the maximum traveling salesman problem. *Information Processing Letters*, 67:125–130, 1998.

[67] Rafael Hassin and Shlomi Rubinstein. Better approximations for max TSP. *Information Processing Letters*, 75:181–186, 2000.

[68] Refael Hassin and Nimrod Meggido. Approximation algorithms for hitting objects with straight lines. *Discrete Applied Mathematics*, 30:29–42, 1991.

[69] Stefan Hertel and Kurt Mehlhorn. Fast triangulation of simple polygons. In *Proceedings of the Fourth International Conference on the Foundations of Computation Theory*, volume 158 of *LNCS*, pages 207–218. Springer-Verlag, LNCS 158, 1983.

[70] David Hilbert. *The Foundations of Geometry*, chapter IV. Open Court Publishing Company, 1902. Translated by Edgar J. Townsend.

[71] Dorit Hochbaum. Approximation algorihtms for the set covering and vertex cover problems. *SIAM Journal on Computing*, 11(3):555–556, 1982.

[72] Dorit S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, 1997.

[73] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM Journal on Computing*, 2:225–231, 1973.

[74] L.J. Hubert and F.B. Baker. Applications of combinatorial programming to data analysis: the traveling salesman and related problems. *Pyschometrika*, 43:81–91, 1978.

[75] Hiroshi Imai and Takao Asano. Dynamic orthogonal segment intersection search. *Journal of Algorithms*, 8(1):1–18, 1987.

[76] David S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

[77] Donald B. Johnson. *Algorithms for Shortest Paths*. PhD thesis, Cornell University, Ithica, NY, 1973.

[78] L. R. Ford Jr. and D. R. Fulkerson. *Flows in networks*. Princeton University Press, 1962.

[79] Simon Kahan and Jack Snoeyink. On the bit complexity of minimum link paths; superquadratic algorithms for problems solvable in linear time. In *Proceedings of the Twelfth Annual Symposium on Computational Geometry*, pages 151–158. ACM Press, 1996.

[80] David R. Karger, 2004. Personal Communication.

[81] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[82] Yan Ke. An efficient algorithm for link-distance problems. In *Proceedings of the 5th annual Symposium on Computational Geometry*, pages 69–78, New York, NY, USA, 1989. ACM Press.

[83] J. Mark Keil. *Decomposing a Polygon into Simpler Components*. PhD thesis, University of Toronto, Toronto, ON, 1983. Report 163/83.

[84] J. Mark Keil. Decomposing a polygon into simpler components. *SIAM Journal on Computing*, 14(4):799–817, November 1985.

[85] J. Mark Keil. Polygon decomposition. In Jörg-Rüdiger Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, chapter 11. Elsevier Science Pub Co, 2000.

[86] Victor Klee. Can the measure of $\cup_1^n [a_i, b_i]$ be computed in less than $O(n \log n)$ steps? *American Mathematical Monthly*, 84(4):284–285, April 1977.

[87] Krasimir Kolarov and Bernhard Roth. On the number of links and placement of telescoping manipulators in an environment with obstacles. In *Interational Conference on Advanced Robotics*, 1991.

[88] Dénis König. Graphok és matrixok. *Matematikai és Fizikai Lápok*, 38:116–119, 1931. (in Hungarian).

[89] S. Rao Kosaraju, James K. Park, and Clifford Stein. Long tours and short superstrings. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 166–177, 1994.

[90] A. V. Kostochka and A. I. Serdyukov. Polynomial algorithms with the estimated 3/4 and 5/6 for the traveling salesman problem of the maximum (in russian). *Upravlyaemye Sistemy*, 26:55–59, 1985.

[91] Evangelos Kranakis, Danny Krizanc, and Lambert Meertens. Link length of rectilinear hamiltonian tours in grids. *Ars Combinatoria*, 38:177–192, 1994.

[92] Eugene L. Lawler, Jan K. Lenstra, A.H.G. Rinooy Kan, and David B. Shmoys, editors. *The Traveling Salesman Problem*. John Wiley and Sons, 1985.

[93] C. Y. Lee. An algorithm for path connections and its applications. *IRE Transactions on Electronic Computers*, EC-10:346–365, September 1961.

[94] D.T. Lee, C.D. Yang, and C.K. Wong. On bends and distances of paths among obstacles in two-layer interconnection model. *IEEE Transactions on Computers*, 43(6):711–724, 1994.

[95] D.T. Lee, C.D. Yang, and C.K. Wong. Rectilinear paths among rectilinear obstacles. *Discrete Applied Mathematics*, 70(3):185–215, 1996.

[96] D.T. Lee, C.D. Yang, and C.K. Wong. Finding rectilinear paths among obstacles in a two-layer interconnection model. *International Journal of Computational Geometry and Applications*, 7(6):581–598, 1997.

[97] Jan van Leeuwen and Derick Wood. The measure problem for rectangular ranges in d-space. *Journal of Algorithms*, 2:282–300, 1981.

[98] William Lenhart, Richard Pollack, Jörg-Rüdiger Sack, Raimund Seidel, Micha Sharir, Subhash Suri, Godfried T. Toussaint, Sue Whitesides, and Chee-Keng Yap. Computing the link center of a simple polygon. *Discrete and Computational Geometry*, 3:281–293, 1988.

[99] N. J. Lennes. Theorems on the simple finite polygon and polyhedron. *American Journal of Mathematics*, 33(1):37–62, January 1911.

[100] Jan K. Lenstra and A.H.G. Rinnooy Kan. Some simple applications of the travelling salesman problem. *Operations Research Quarterly*, 26:717–733, 1975.

[101] Christos Levcopoulos and Andrzej Lingas. Bounds on the length of convex partitions of polygons. In *Proceedings of the Fourth Conference on the Foundations of Software Technology and Theoretical Computer Science*, volume 181 of *LNCS*, pages 279–295. Springer-Verlag, LNCS 181, 1984.

[102] Andrzej Lingas. The power of non-rectilinear holes. In *Proceedings of the Ninth International Colloquium on Automata, Languages, and Programming*, volume 140 of *LNCS*, pages 369–383. Springer-Verlag, LNCS 140, 1982.

[103] Andrzej Lingas, Anil Maheshwari, and Jörg-Rüdiger Sack. Optimal parallel algorithms for rectilinear link-distance problems. *Algorithmica*, 14:261–289, 1995.

[104] Andrzej Lingas, Anil Maheshwari, and Jörg-Rüdiger Sack. Optimal parallel algorithms for rectilinear link-distance problems. *Algorithmica*, 14(3):261–289, 1995.

[105] Andrzej Lingas, Ron Y. Pinter, Ronald L. Rivest, and Adi Shamir. Minimum edge length partitioning of rectlinear polygons. In *Proceedings of the 20th Annual Allerton Conference on Communcation, Control, and Computing*, pages 53–63, 1982.

[106] W. T. Liou, Jimmy J. M. Tan, and Richard C. T. Lee. Minimum partitioning simple rectilinear polygons in $O(n \log \log n)$-time. In *Proceedings of the 1989 ACM Symposium on Computational Geometry*, pages 344–353, 1989.

[107] Witold Lipski Jr. Finding a Manhattan path and related problems. *Networks*, 13:399–409, 1983.

[108] Witold Lipski Jr. An $O(n \log n)$ Manhattan path algorithm. *Information Processing Letters*, 19:99–102, 1984.

[109] Witold Lipski Jr., Elena Lodi, Fabrizio Luccio, C. Mugnai, and Linda Pagli. On two-dimensional data organization II. *Fundamenta Informaticae*, 2:245–260, 1979.

[110] László Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13:383–390, 1975.

[111] Carsten Lund and Mihalis Yannakakis. On the hardness of approximating minimization problems. *Journal of the ACM*, 41:960–981, September 1994.

[112] Anil Maheshwari, Jörg-Rüdiger Sack, and Hristo N. Djidjev. Link distance problems. In Jörg-Rüdiger Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, chapter 12, pages 519–558. Elsevier Science Pub Co, 2000.

[113] W.T. McCormick, P.J. Schweitzer, and T.W. White. Problem decomposition and data reorganization by a clustering technique. *Operations Research*, 20:993–1009, 1972.

[114] Edward M. McCreight. Efficient algorithms for enumerating intersecting intervals and rectangles. Report CSL-80-9, Xerox Palo Alto Research Center, Palo Alto, CA, 1980.

[115] Kenneth M. McDonald and Joseph G. Peters. Smallest paths in simple rectilinear polygons. *IEEE Transactions on Computer-Aided Design*, 11(7):864–875, 1992.

[116] Nimrod Meggido and Arie Tamir. On the complexity of locating linear facilities in the plane. *Operations Research Letters*, 1:194–197, 1982.

[117] Koichi Mikami and Kinya Tabuchi. A computer program for optimal routing of printed circuit conductors. In *Proceedings of the International Federation for Information Processing Congress 68*, pages 1475–1478, 1968.

[118] Joseph S. B. Mitchell. *Planning Shortest Paths*. PhD thesis, Stanford University, Stanford, CA, 1986.

[119] Joseph S. B. Mitchell. Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, k-MST, and related problems. *SIAM Journal on Computing*, 28:1298–1309, 1999.

[120] Joseph S. B. Mitchell. Geometric shortest paths and network optimization. In Jörg-Rüdiger Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, chapter 15, pages 633–641. Elsevier Science Pub Co, 2000.

[121] Joseph S. B. Mitchell, 2003. Personal Communication.

[122] Joseph S. B. Mitchell. Shortest paths and networks. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry (2nd Edition)*, chapter 27, pages 607–641. Chapman & Hall/CRC, Boca Raton, FL, 2004.

[123] Joseph S. B. Mitchell, Christine D. Piatko, and Esther M. Arkin. Computing a shortest k-link path in a polygon. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pages 573–582. IEEE, 1992.

[124] Joseph S. B. Mitchell, Günter Rote, and Gerhard J. Woeginger. Minimum-link paths among obstacles in the plane. *Algorithmica*, 8:431–459, 1992.

[125] Bernard M. E. Moret. The ice rink problem. In *Proceedings of the First Workshop on Algorithm Engineering*, Venice, Italy, 1997.

[126] Bengt J. Nilsson and Sven Schuierer. Computing the rectiinear link diameter of a polygon. In *Methods, Algorithms, and Applications: Proceedings of the International Workshop on Computational Geometry*, volume 553 of *LNCS*, pages 203–215. Springer-Verlag, LNCS 553, 1991.

[127] Bengt J. Nilsson and Sven Schuierer. An optimal algorithm for the rectilinear link center of a rectilinear polygon. *Computational Geometry: Theory and Applications*, 6(3):169–194, 1996.

[128] Tatsuo Ohtsuki. Minimum dissection of rectilinear regions. In *Proceedings of the IEEE International Conference on Circuits and Systems*, pages 1210–1213, 1982.

[129] Tatsuo Ohtsuki and Masao Sato. Gridless routers for two-layer interconnection. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 76–78, 1984.

[130] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, 1998. Section 4.4.

[131] Joseph O'Rourke. *Computational Geometry in C*, chapter 2. Cambridge University Press, 1998.

[132] John K. Ousterhout. Corner stitching: A data-structuring technique for VLSI layout tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, CAD-3(1):87–100, January 1984.

[133] Mark H. Overmars. Geometric data structures for computer graphics: an overview. *Theoretical Foundations of Computer Graphics and CAD. NATO ASI Series F*, 40:21–49, 1988.

[134] Mark H. Overmars. Computational geometry and its application to computer graphics. In *Advances in Computer Graphics*, pages 75–107, 1989.

[135] Mark H. Overmars and Chee-Keng Yap. New upper bounds in Klee's measure problem. *SIAM Journal on Computing*, 20(6):1034–1045, December 1991.

[136] Christos H. Papadimitriou. The euclidean traveling salesman problem is NP-complete. *Theoretical Computer Science*, 4:237–244, 1977.

[137] Christos H. Papadimitriou and Mihalis Yannakakis. The traveling salesman problem with distances one and two. *Mathematics of Operations Research*, 18(1):1–11, 1993.

[138] Mike S. Paterson and F. Frances Yao. Optimal binary space partitions for orthogonal objects. *Journal of Algorithms*, 13:99–113, 1992.

[139] Christine D. Piatko. *Geometric Bicriteria Optimal Path Problems*. PhD thesis, Cornell University, Ithica, NY, 1993.

[140] Valentin Polishchuk and Joseph S. B. Mitchell. k-link rectilinear shortest paths among recilinear obstacles in the plane. In *The 17th Canadian Conference on Computational Geometry*, pages 98–101, 2005.

[141] John H. Reif and James A. Storer. Minimizing turns for the discrete movement in the interior of a polygon. *IEEE Journal of Robotics and Automation*, 3:182–193, 1987.

[142] Günter Rote, 2005. Personal Communication.

[143] Shmuel Safra and Oded Schwartz. On the complexity of approximating TSP with neighborhoods and related problems. In *The 11th Annual European Symposium on Algorithms*, pages 446–458, Budapest, Hungary, 2003.

[144] Sartaj Sahni and Teofilo Gonzales. P-complete approximation problems. *Journal of the ACM*, 23(3):555–565, July 1976.

[145] Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, 1986.

[146] Masao Sato, Jiro Sakanaka, and Tatsuo Ohtsuki. A fast line-search algorithm. Technical Report CAS 85-154, The Institute of Electronics and Communications Engineers (IECE) of Japan, 1986. (in Japanese).

[147] Masao Sato, Jiro Sakanaka, and Tatsuo Ohtsuki. A fast line-search method based on a tile plane. In *Proceedings of the IEEE International Conference on Circuits and Systems*, pages 588–591, 1987.

[148] Petr Slavík. A tight analysis of the greedy algorithm for set cover. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, pages 435 – 441, 1996.

[149] Michael G. Smith. Avalanche transceiver recovery skills (online article), 1998. http://gearworld.com/TipsAndTraps/AvalancheRescue/.

[150] Clifford Stein and David P. Wagner. Approximation algorithms for the minimum bends traveling salesman problem. In *Proceedings of The Eighth International Integer Programming and Combinatorial Optimization Conference*, pages 406–421. Springer-Verlag, LNCS 2081, 2001.

[151] Subhash Suri. A linear time algorithm with minimum link paths inside a simple polygon. *Comput. Vision Graph. Image Process.*, 35(1):99–110, 1986.

[152] Subhash Suri. *Minimum Link Paths in Polygons and Related Problems*. PhD thesis, Johns Hopkins University, Baltimore, MD, 1987.

[153] Subhash Suri. On some link distance problems in a simple polygon. *IEEE Transactions on Robotics and Automation*, 6(1):108–113, 1990.

[154] Kei Suzuki, Tatsuo Ohtsuki, and Masao Sato. A gridless router: Software and hardware implementations. In *VLSI '87*, pages 153–163, 1987.

[155] Roberto T. Tamassia. On embedding a graph in the grid with minimum number of bends. *SIAM Journal of Computing*, 1986.

[156] Robert E. Tarjan and C. J. Van Wyk. An $O(n \log \log n)$-time algorithm for triangulating a simple polygon. *SIAM Journal on Computing*, 17(1):143–178, February 1988.

[157] Vijay K. Vaishnavi. Computing point enclosures. *IEEE Transactions on Computers*, C-31:22–29, 1982.

[158] Vijay K. Vaishnavi and Derick Wood. Rectilinear line segment intersection, layered segment trees, and dynamization. *Journal of Algorithms*, 3:160–176, 1982.

[159] V. N. Vapnik and A. Ya. Červonenekis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16:264–280, 1971.

[160] C.D. Yang, D.T. Lee, and C.K. Wong. On bends and lengths of rectilinear paths: a graph-theoretic approach. *Internat. J. Comp. Geom. Appl.*, 2:61–74, 1992.

[161] C.D. Yang, D.T. Lee, and C.K. Wong. Rectilinear path problems among rectilinear obstacles revisited. *SIAM Journal on Computing*, 24:457–472, 1992.

[162] C.D. Yang, D.T. Lee, and C.K. Wong. The smallest pair of noncrossing paths in a rectilinear polygon. *IEEE Transactions on Computers*, 46(8):930–941, 1997.

Appendix

Source Code

The proceeding sections of this Appendix contain the source code that was used to implement this algorithm. This following list describe the contents of each file.

- **datatypes.h** lists the datatypes used in the implementation.
- **segtree.c** contains all the functions whic operate on one dimensional and two dimensional segment trees.
- **subface.c** contains functions which operate on subfaces, and lists of subfaces.
- **space.c** contains functions which operate on the space itself.
- **block.c** contains functions which operate on the blocks, and lists of blocks.
- **generate.c** contains functions which generate events. It also contains the main algorthm.
- **queue.c** contains function which handle the priority queue.
- **sweepplane.c** contains functions which handle the sweep plane.

Listings

Source code for :datatypes.h

```
1  // datatypes.h
       David Wagner
2  // Holds the datatype for mbp.c, mbp.h
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7  #include <time.h>
8
9  #define FALSE 0
10 #define TRUE 1
11
12 #define EmptyBlock 0
13 #define ObstacleFace 1
14 #define OutgoingPath 2
15
16 #define XAxis 0
17 #define YAxis 1
18 #define ZAxis 2
19 #define NoAxis 3
20
21 #define NEG 0
22 #define POS 1
23
24 // These are the names of the files which store the
25 // 1D and 2D segment trees
26
27 #define FILEX   "treex.dat"
28 #define FILEY   "treey.dat"
29 #define FILE2D "tree2D.dat"
30
31 #define NONE 0
32 #define START 1
33 #define FINISH 2
34 #define INSIDE 3
35 #define OUTSIDE 4
36 #define OBSTACLE 5
37
38 #define NOPATH -1
39
40 #define NOTICE 1
41 #define WARNING 2
```

```
42 #define ERROR 3
43 #define SEVERE 4
44 #define CRITICAL 5
45
46 #define QUERYTOHALT 4
47 #define HALT 5
48
49 #define WALLCUBE 'W'
50 #define FACECUBE 'F'
51 #define EITHERCUBE 'E'
52
53
54 #define PRINT_REMOVE_EVENT FALSE
55 #define PRINT_INSERT_EVENT FALSE
56 #define PRINT_SPLITTING_ON_FACE FALSE
57 #define PRINT_ALL_FACES_IN_SPACE FALSE
58 #define PRINT_SPACE_BOUNDS FALSE
59 #define PRINT_NUM_FACES_IN_SPACE FALSE
60 #define PRINT_BLOCK_SET FALSE
61 #define PRINT_OLD_BLOCK FALSE
62 #define PRINT_GENERATING_PATHS FALSE
63 #define PRINT_PATH_FOUND FALSE
64 #define PRINT_FOUND_NEIGHBORING FALSE
65 #define PRINT_ADDING_CUBE FALSE
66
67 #define PRINT_NEW_SWEEP_PLANE FALSE
68
69 #define PRINT_INSERTED_SUBFACE FALSE
70
71 #define PRINT_INSERTED_CUBE FALSE
72
73 #define PRINT_MY_SPACE FALSE
74 #define PRINT_INTERACTIVE FALSE
75
76 #define PRINT_EVENT_COUNT TRUE
77
78 typedef int EventType;
79 typedef int Direction;
80 typedef int Num;
81 typedef char Boolean;
82 typedef int Axis;
83 typedef char Sign;
84 typedef int PointType;
85 typedef int BlockType;
86 typedef int FaceType;
```

```
87  typedef int Label;
88
89
90  // The Segment Tree Datatypes
91
92  // A single node in a segment tree
93
94  typedef
95  struct SEGTREENODE {
96    Num l,r; // The range of this node  l<=r
97    struct SEGTREENODE * left, * right;  // the children
98    Boolean data, subtreedata;
99    PointType pt;
100 } SegTreeNode;
101
102
103 // A single node in a two dimensional segment tree
104
105 typedef
106 struct SEGTREENODE2D {
107   Num b, t; // The range of this node: b<=t
108   struct SEGTREENODE2D * left, * right;  // the children
109   SegTreeNode * perptree, * projtree;
110 } SegTreeNode2D;
111
112 // A set of one dimensional segment tree nodes
113 // This is maintained as a circular doubly linked list
114
115 typedef
116 struct NODESET {
117   SegTreeNode * node;
118   struct NODESET * next, * prev;
119 } NodeSet;
120
121 // A set of two dimensional segment tree nodes
122
123 typedef
124 struct NODE2DSET {
125   SegTreeNode2D * node;
126   struct NODE2DSET * next, * prev;
127 } NodeSet2D;
128
129
130 // The Event Datatype
131
```

```
132 typedef
133 struct EVENT {
134   int   benddist;
135 //   Direction   dir;
136   Axis axis;
137   Sign sign;
138   EventType eventtype;
139   struct BLOCK * block;
140   SegTreeNode * data;
141   Axis stripeAxis; // which way, X or Y, to stripe the data
142         // after removal from the queue
143   Num   coord;
144   Num   left, right, bottom, top; // stripe coords after
          removal from the queue
145
146   Label label;
147
148 } Event;
149
150 // The Priority Queue
151
152 typedef
153 struct PQUEUE {
154   int length;
155   int allocated;
156
157   Event ** heap;
158 } PQueue;
159
160
161 //    Coordinate system with dimension and sign
162 //
163 //        Y
164 //        |    Z
165 //        |   /
166 //        |  /
167 //        |/_____ X
168
169
170 // A subface of an obstacle face
171
172 typedef
173 struct SUBFACE {
174   Axis axis;
175   // Which side of the face is the outside?
```

```
176     Boolean sign;
177     // Does this Face bound the Finish or Start?
178     FaceType facetype;
179     // These bounds are with reference to the sweep direction
180     Num top, bottom, left, right, front, back;
181     // These bounds are an absolute direction as [dimension][
            sign]
182     // It should be the case that bound[axis][0]==bound[axis
            ][1]==coord
183     Num bound[3][2];
184     Num coord;
185     int lastsweep[3][2];
186   } SubFace;
187
188
189   // A set of obstacle subfaces
190
191   typedef
192   struct SUBFACESET {
193     SubFace * face;
194     int count;
195     struct SUBFACESET * next, * prev;
196   } SubFaceSet;
197
198
199   // A Block in the BSP decomposition
200
201   typedef
202   struct BLOCK {
203     int minbenddist;
204     // Inside or Outside?  Start or Finish?
205     BlockType blocktype;
206     // These bounds are with reference to the current sweep
            direction
207     // They may rotate to any cardinal direction
208     Num top, bottom, left, right, front, back;
209     // These bounds are an absolute direction as [dimension][
            sign]
210     Num bound[3][2];
211     SubFaceSet * faceNeighbors[3][2];
212     struct BLOCKSET * blockNeighbors [3][2];
213     Label label;
214     // lastsweep contains the benddist of the last sweep into
            which
215     // this block was inserted
```

```
216   int lastsweep[3][2];
217 } Block;
218
219
220 // A set of blocks
221
222 typedef
223 struct BLOCKSET {
224   Block * block;
225   int count; // Only assigned in the head block
226   Block * startblock, * finishblock; // Only assigned in the
          head block
227   struct BLOCKSET * next, * prev;
228 } BlockSet;
229
230
231 typedef
232 struct POINT {
233   Num x, y, z;
234 } Point;
235
236
237 typedef
238 struct SPACE {
239
240   // A list of faces parallel to each dimension and
241   // facing the positive and negative directions
242   SubFaceSet * faces[3][2];
243   int facecount[3][2];
244
245   Point * start, * finish;
246
247   // These bounds are with reference to the current direction
248   // They may rotate to any cardinal direction
249   Num top, bottom, left, right, front, back;
250   // These bounds are an absolute direction as [dimension][
          sign]
251   Num bound[3][2];
252
253   SegTreeNode2D * inside[3][2];
254
255   Label label;
256
257 } Space;
258
```

```
259
260 // A node of the BSP Tree
261
262 typedef
263 struct BSPNODE {
264   Block * block;
265
266   struct BSPNODE * posNode, * negNode, * parentNode;
267
268   SubFace * splittingFace;
269   Space * space;
270   Axis splitaxis;
271
272   int subtreenodecount, subtreeleafcount;
273
274 } BSPNode;
275
276
277 // The Sweep Plane
278
279 typedef
280 struct SWEEPPLANE {
281   SegTreeNode2D * data;
282   Axis axis;
283   Sign sign;
284   int benddist;
285 } SweepPlane;
```

Listing 8.1: datatypes.h

Source code for :segtree.c

```
1  // segtree.c
       David Wagner
2  // Operations on segment trees
3
4  #include "include.h"
5  #include "segtree.h"
6  #include "bounds.h"
7
8
9  ////////////////// Global Variables ////////////////////////
10
11 SegTreeNode * mySegTree;
12 SegTreeNode2D * mySegTree2D;
13
14
15 /////////////// Basic Operations ////////////////////////
16
17 Num min(Num a, Num b)
18 {
19   return(a<b?a:b);
20 }
21
22 Num max(Num a, Num b)
23 {
24   return(a>b?a:b);
25 }
26
27 /////////////// Operations on Nodes ////////////////////////
28
29 SegTreeNode * newNode(Num l, Num r)
30 {
31   SegTreeNode * myNode = malloc (sizeof (SegTreeNode));
32
33   myNode->l = l; myNode->r = r;
34   myNode->left = myNode->right = NULL;
35   myNode->data = myNode->subtreedata = FALSE;
36
37   checkLRBounds(l, r);
38
39   return myNode;
40 }
41
```

```
42 SegTreeNode2D * new2DNode(Num b, Num t)
43 {
44   SegTreeNode2D * myNode = malloc (sizeof (SegTreeNode2D));
45
46   myNode->t = t; myNode->b = b;
47   myNode->left = myNode->right = NULL;
48   myNode->perptree = myNode->projtree = NULL;
49
50   checkBTBounds(b, t);
51
52   return myNode;
53 }
54
55 Num getNodeLeft(SegTreeNode * node) { return node->l; }
56 Num getNodeRight(SegTreeNode * node) { return node->r; }
57
58 Num getNodeLeft2D(SegTreeNode2D * node) { return node->
       perptree->l; }
59 Num getNodeRight2D(SegTreeNode2D * node) { return node->
       perptree->r; }
60 Num getNodeBottom2D(SegTreeNode2D * node) { return node->t; }
61 Num getNodeTop2D(SegTreeNode2D * node) { return node->b; }
62
63 Boolean nodeWithinRange(SegTreeNode * node, Num l, Num r)
64 {
65   return (l <= node->l && r >= node->r);
66 }
67
68 Boolean nodeOverlapsRange(SegTreeNode * node, Num l, Num r)
69 {
70   return !(l > node->r || r < node->l);
71 }
72
73 Boolean rangeWithinNode(SegTreeNode * node, Num l, Num r)
74 {
75   return (l >= node->l && r <= node->r);
76 }
77
78 Boolean nodeWithinRange2D(SegTreeNode2D * node, Num b, Num t)
79 {
80   return (t >= node->t && b <= node->b);
81 }
82
83 Boolean nodeOverlapsRange2D(SegTreeNode2D * node, Num b, Num t
       )
```

```
84  {
85    return !(t < node->b || b > node->t);
86  }
87
88  Boolean rangeWithinNode2D(SegTreeNode2D * node, Num b, Num t)
89  {
90    return (t <= node->t && b >= node->b);
91  }
92
93  void maintainNode(SegTreeNode * node)
94  {
95    if (node->left && node->left->data == TRUE &&
96      node->right && node->right->data == TRUE) {
97      node->data = TRUE;
98      ClearChildrenSubtrees(node);
99    }
100
101   node->subtreedata = node->data;
102   if (node->left) node->subtreedata |= node->left->subtreedata
        ;
103   if (node->right) node->subtreedata |= node->right->
        subtreedata;
104
105   if (node->left && node->left->data == FALSE &&
106     node->left->left == NULL && node->left->right == NULL)
107       ClearLeftSubtree(node);
108   if (node->right && node->right->data == FALSE &&
109     node->right->left == NULL && node->right->right == NULL)
110       ClearRightSubtree(node);
111
112 }
113
114 void maintainTree(SegTreeNode * root)
115 {
116   if (root == NULL) return;
117
118   maintainTree(root->left);
119   maintainTree(root->right);
120
121   maintainNode(root);
122 }
123
124 void maintainNode2D(SegTreeNode2D * node)
125 {
126   ClearSubtree(node->projtree);
```

```
127   node->projtree = CopyTree(node->perptree);
128   if(node->left) UnionTrees(node->projtree, node->left->
          projtree);
129   if(node->right) UnionTrees(node->projtree, node->right->
          projtree);
130 }
131
132 ////////////////// Operations creating new Trees
        ///////////////////////
133
134 SegTreeNode * newSegTree(Num l, Num r)
135 {
136   if (r<l) return(NULL);
137
138   return newNode(l,r);
139 }
140
141 SegTreeNode2D * newSegTree2D(Num l, Num r, Num b, Num t)
142 {
143   SegTreeNode2D * myTree;
144
145   if (!checkLRBTBounds(l,r,b,t)) return(NULL);
146
147   myTree = new2DNode(b,t);
148   myTree->perptree = newSegTree(l,r);
149   myTree->projtree = newSegTree(l,r);
150
151   if (b<t) {
152     myTree->left = newSegTree2D(l, r, b, (t+b)/2);
153     myTree->right = newSegTree2D(l, r, (t+b)/2+1, t);
154   }
155
156   return myTree;
157 }
158
159 void clearSegTree2D(SegTreeNode2D * T)
160 {
161   if (T==NULL) return;
162
163   ClearSubtree(T->perptree);
164   ClearSubtree(T->projtree);
165
166   clearSegTree2D(T->left);
167   clearSegTree2D(T->right);
168
```

```
169   free(T);
170 }
171
172
173 // Create the children of this node, if they do not exist,
174 // unless this node is intended to be a leaf
175
176 void ensureChildrenExist(SegTreeNode * root)
177 {
178   if (root==NULL) return;
179   if (root->l >= root->r) return;  // The node is a leaf
180
181   if (root->left == NULL) root->left =
182     newSegTree(root->l, (root->l + root->r) / 2);
183   if (root->right == NULL) root->right =
184     newSegTree((root->l + root->r) / 2 + 1, root->r);
185
186 }
187
188
189 /////////////// Operations on NodeSets ////////////////////
190
191 // Take a node, and return a set of nodes containing
192 // that node as the only element.
193
194 NodeSet * UnitNodeSet(SegTreeNode * node)
195 {
196   NodeSet * set = (NodeSet *) malloc(sizeof(NodeSet));
197
198   set->next = set->prev = set;
199   set->node = node;
200
201   return set;
202 }
203
204 // Take a 2D node, and return a set of nodes containing
205 // that node as the only element.
206
207 NodeSet2D * UnitNodeSet2D(SegTreeNode2D * node)
208 {
209   NodeSet2D * set = (NodeSet2D *) malloc(sizeof(NodeSet2D));
210
211   set->next = set->prev = set;
212   set->node = node;
213
```

```
214     return set;
215 }
216
217
218 // Take two sets of nodes and union them together
219 // Both original sets are destroyed
220 // Either input pointer can be used to reference the new set
221 // O(1)
222
223 void UnionNodeSets(NodeSet * set1, NodeSet * set2)
224 {
225   NodeSet * temp;
226
227   if (set1==NULL) {set1 = set2; return;}
228   if (set2==NULL) {set2 = set1; return;}
229
230   set1->prev->next = set2;
231   set2->prev->next = set1;
232   temp = set1->prev;
233   set1->prev = set2->prev;
234   set2->prev = temp;
235 }
236
237 // Take two sets of 2D nodes and union them together
238 // Both original sets are destroyed
239 // Either input pointer can be used to reference the new set
240 // O(1)
241
242 void UnionNodeSets2D(NodeSet2D * set1, NodeSet2D * set2)
243 {
244   NodeSet2D * temp;
245
246   if (set1==NULL) {set1 = set2; return;}
247   if (set2==NULL) {set2 = set1; return;}
248
249   set1->prev->next = set2;
250   set2->prev->next = set1;
251   temp = set1->prev;
252   set1->prev = set2->prev;
253   set2->prev = temp;
254 }
255
256 // Insert the given node into the existing NodeSet
257
258 void InsertNodeSet(NodeSet * set, SegTreeNode * node)
```

```
259 {
260   NodeSet * newSet = UnitNodeSet(node);
261   UnionNodeSets(set, newSet);
262 }
263
264
265 //////////////// Operations on Segment Trees
      //////////////////
266
267 // Return the set of canonical nodes within the given tree
268 // representing the given range
269 // O(lgn)
270
271 NodeSet * CanonicalNodes(SegTreeNode * root, Num l, Num r)
272 {
273   NodeSet * leftSet, * rightSet;
274
275   if (root==NULL) return NULL;
276   if (l>r) return NULL;
277
278   // If the input range does not overlap the range of this
        node,
279   // then return NULL.
280   if (!nodeOverlapsRange(root, l, r)) return(NULL);
281
282   // If the input range entirely encompasses the range
283   // of this node, then return this node as a unit nodeset
284   if (nodeWithinRange(root, l, r)) return(UnitNodeSet(root));
285
286   ensureChildrenExist(root);
287
288   leftSet = CanonicalNodes(root->left, l, r);
289   rightSet = CanonicalNodes(root->left, l, r);
290   UnionNodeSets(leftSet, rightSet);
291   return leftSet;
292 }
293
294 // Return the set of all ancestor nodes of all canonical nodes
295 // representing the given range.
296 // O(lgn)
297
298 NodeSet * CanonicalAncNodes(SegTreeNode * root, Num l, Num r)
299 {
300   NodeSet *mySet, * leftSet, * rightSet;
301
```

```
302     if (root==NULL) return NULL;
303     if (l>r) return NULL;
304
305     // If the input range does not overlap the range of this
            node,
306     // then there are no ancestors in this subtree.  Return NULL
            .
307     if (!nodeOverlapsRange(root, l, r)) return(NULL);
308
309     // If the input range entirely encompasses the range
310     // of this node, then there are no ancestors in this
311     // subtree.  Return NULL.
312     if (nodeWithinRange(root, l, r)) return(NULL);
313
314     ensureChildrenExist(root);
315
316     mySet = UnitNodeSet(root);
317     leftSet = CanonicalNodes(root->left, l, r);
318     rightSet = CanonicalNodes(root->left, l, r);
319     UnionNodeSets(leftSet, rightSet);
320     UnionNodeSets(leftSet, mySet);
321
322     return leftSet;
323
324 }
325
326 // Return the set of all decendant nodes of all canonical
        nodes
327 // representing the given range
328 // O(n)
329
330 NodeSet * CanonicalDecNodes(SegTreeNode * root, Num l, Num r)
331 {
332   NodeSet * mySet, * leftSet, * rightSet;
333
334     if (root==NULL) return NULL;
335     if (l>r) return NULL;
336
337     // If the input range does not overlap the range of this
            node,
338     // then return NULL.
339     if (!nodeOverlapsRange(root, l, r)) return(NULL);
340
341     leftSet = CanonicalDecNodes(root->left, l, r);
342     rightSet = CanonicalDecNodes(root->right, l, r);
```

```
343
344   UnionNodeSets(leftSet, rightSet);
345
346   // If the input range entirely encompasses the range
347   // of this node, then return me in addition to my decendants
          .
348
349   if (nodeWithinRange(root, l, r)) {
350     mySet = UnitNodeSet(root);
351     UnionNodeSets(leftSet, mySet);
352   }
353
354   return(leftSet);
355 }
356
357 // Return a new copy of the given tree
358 // O(|T|)
359
360 SegTreeNode * CopyTree(SegTreeNode * T)
361 {
362   SegTreeNode * newTree;
363
364   if (T==NULL) return NULL;
365
366   newTree=newNode(T->l, T->r);
367
368   newTree->left=CopyTree(T->left);
369   newTree->right=CopyTree(T->right);
370   newTree->data=T->data;
371   newTree->subtreedata=T->subtreedata;
372
373   return newTree;
374 }
375
376 // Union the second tree into the first
377 // O(|T2|)
378
379 void UnionTrees(SegTreeNode * T1, SegTreeNode * T2)
380 {
381   if (T2 == NULL) return;
382   if (T1 == NULL) {
383     genError(ERROR, "UnionTrees", "T1 NULL");
384     return;
385   }
386   if (T1->l != T2->l) {
```

```
387       genErrorIntInt(ERROR, "UnionTrees", "Left %d != %d",
388         T1->l, T2->l);
389       return;
390     }
391
392     if (T1->r != T2->r) {
393       genErrorIntInt(ERROR, "UnionTrees", "Right %d != %d",
394         T1->r, T2->r);
395       return;
396     }
397     T1->data = T1->data || T2->data;
398     T1->subtreedata = T1->subtreedata || T2->subtreedata;
399
400     if(T1->data) {
401       ClearChildrenSubtrees(T1);
402       return;
403     }
404
405     if(T2->left || T2->right) {
406       ensureChildrenExist(T1);
407       UnionTrees(T1->left, T2->left);
408       UnionTrees(T1->right, T2->right);
409     }
410
411     maintainNode(T1);
412   }
413
414   // Clears the subtree rooted at the given node
415   // O(D)
416
417   void ClearSubtree(SegTreeNode * T)
418   {
419     if(T == NULL) return;
420     ClearSubtree(T->left);
421     ClearSubtree(T->right);
422     free(T);
423   }
424
425   // Clears the subtree rooted at the left child of the given
        node
426   // O(D)
427
428   void ClearLeftSubtree(SegTreeNode * T)
429   {
430     if(T == NULL) return;
```

```
431     ClearSubtree(T->left);
432     T->left = NULL;
433   }
434
435   // Clears the subtree rooted at the right child of the given
        node
436   // O(D)
437
438   void ClearRightSubtree(SegTreeNode * T)
439   {
440     if(T == NULL) return;
441     ClearSubtree(T->right);
442     T->right = NULL;
443   }
444
445   // Clears the subtrees rooted at both children of the given
        node
446   // O(D)
447
448   void ClearChildrenSubtrees(SegTreeNode * T)
449   {
450     if(T == NULL) return;
451     ClearLeftSubtree(T);
452     ClearRightSubtree(T);
453   }
454
455   // Removes all data outside the specified range
456   // O(lgn + D)
457
458   void TruncateTree(SegTreeNode * T, Num l, Num r)
459   {
460     if(T == NULL) return;
461     if (l>r) return;
462
463     if(nodeWithinRange(T, l, r)) return;
464
465     if(T->data == TRUE){
466       ensureChildrenExist(T);
467       T->left->data = TRUE;
468       T->right->data = TRUE;
469       T->data = FALSE;
470     }
471
472     if(T->left && !nodeOverlapsRange(T->left, l, r))
          ClearLeftSubtree(T);
```

```
473     if(T->right && !nodeOverlapsRange(T->right, l, r))
            ClearRightSubtree(T);
474
475     TruncateTree(T->left, l, r);
476     TruncateTree(T->right, l, r);
477
478     maintainNode(T);
479   }
480
481   // Insert the specified range into the segment tree
482
483   void InsertRange(SegTreeNode * T, Num l, Num r)
484   {
485     if(T == NULL) return;
486
487     if(!nodeOverlapsRange(T, l, r)) return;
488     if (l>r) return;
489
490     if(nodeWithinRange(T, l, r)) {
491       T->data = TRUE;
492       ClearChildrenSubtrees(T);
493       maintainNode(T);
494       return;
495     }
496
497     ensureChildrenExist(T);
498
499     InsertRange(T->left, l, r);
500     InsertRange(T->right, l, r);
501
502     maintainNode(T);
503   }
504
505   // Create a new segment tree containing only those nodes which
506   // are needed to represent the given range.
507
508   SegTreeNode * NewCanonicalTree(Num min, Num max, Num l, Num r)
509   {
510     SegTreeNode * node = newNode(min, max);
511     InsertRange(node, l, r);
512
513     return node;
514   }
515
516   // Clear all nodes in the specified range from the subtree
```

```
517 // O(lgn + D)
518
519 void ClearRange(SegTreeNode * T, Num l, Num r)
520 {
521   if(T == NULL) return;
522   if(l>r) return;
523
524   if(!nodeOverlapsRange(T, l, r)) return;
525
526   if(nodeWithinRange(T, l, r)) {
527     ClearChildrenSubtrees(T);
528     T->data == FALSE;
529   }
530
531   if(T->data == TRUE) {
532     ensureChildrenExist(T);
533     if(T->left) { T->left->data = TRUE; maintainNode(T->left);
          }
534     if(T->right) { T->right->data = TRUE; maintainNode(T->
          right); }
535     T->data = FALSE;
536   }
537
538   if(T->left && nodeWithinRange(T->left, l, r))
        ClearLeftSubtree(T);
539   if(T->right && nodeWithinRange(T->right, l, r))
        ClearRightSubtree(T);
540
541   ClearRange(T->left, l, r);
542   ClearRange(T->right, l, r);
543
544   maintainNode(T);
545
546 }
547
548 // Return TRUE if there is any data in T within the indicated
      range
549 // O(lgn)
550
551 Boolean QueryRange(SegTreeNode * T, Num l, Num r)
552 {
553   if (T == NULL) return FALSE;
554   if (l>r) return FALSE;
555
556   if (!nodeOverlapsRange(T, l, r)) return FALSE;
```

```
557
558   if (T->data) return TRUE;
559
560   if (nodeWithinRange(T, l, r)) return (T->subtreedata);
561
562   if (QueryRange(T->left, l, r)) return TRUE;
563   return (QueryRange(T->right, l, r));
564
565 }
566
567 ////////////////////  Operations on 2D segment trees
        ////////////////////
568
569 // Return the set of canonical nodes within the given tree
570 // representing the given range
571 // O(lgn)
572
573 NodeSet2D * CanonicalNodes2D(SegTreeNode2D * root, Num b, Num
        t)
574 {
575   NodeSet2D * leftSet, * rightSet;
576
577   if (root==NULL) return NULL;
578   if (!checkBTBounds(b, t)) return NULL;
579
580   // If the input range does not overlap the range of this
        node,
581   // then return NULL.
582   if (!nodeOverlapsRange2D(root, b, t)) return(NULL);
583
584   // If the input range entirely encompasses the range
585   // of this node, then return this node as a unit nodeset
586   if (nodeWithinRange2D(root, b, t)) return(UnitNodeSet2D(root
        ));
587
588
589   leftSet = CanonicalNodes2D(root->left, b, t);
590   rightSet = CanonicalNodes2D(root->left, b, t);
591   UnionNodeSets2D(leftSet, rightSet);
592   return leftSet;
593 }
594
595 // Return the set of all ancestor nodes of all canonical nodes
596 // representing the given range.
597 // O(lgn)
```

```
598
599 NodeSet2D * CanonicalAncNodes2D(SegTreeNode2D * root, Num b,
        Num t)
600 {
601   NodeSet2D *mySet, * leftSet, * rightSet;
602
603   if (root==NULL) return NULL;
604   if (!checkBTBounds(b, t)) return NULL;
605
606   // If the input range does not overlap the range of this
        node,
607   // then there are no ancestors in this subtree.  Return NULL
        .
608   if (!nodeOverlapsRange2D(root, b, t)) return(NULL);
609
610   // If the input range entirely encompasses the range
611   // of this node, then there are no ancestors in this
612   // subtree.  Return NULL.
613   if (nodeWithinRange2D(root, b, t)) return(NULL);
614
615   mySet = UnitNodeSet2D(root);
616   leftSet = CanonicalNodes2D(root->left, b, t);
617   rightSet = CanonicalNodes2D(root->left, b, t);
618   UnionNodeSets2D(leftSet, rightSet);
619   UnionNodeSets2D(leftSet, mySet);
620
621   return leftSet;
622
623 }
624
625 // Return the set of all decendant nodes of all canonical
        nodes
626 // representing the given range
627 // O(n)
628
629 NodeSet2D * CanonicalDecNodes2D(SegTreeNode2D * root, Num b,
        Num t)
630 {
631   NodeSet2D * mySet, * leftSet, * rightSet;
632
633   if (root==NULL) return NULL;
634   if (!checkBTBounds(b, t)) return NULL;
635
636   // If the input range does not overlap the range of this
        node,
```

```
637     // then return NULL.
638     if (!nodeOverlapsRange2D(root, b, t)) return(NULL);
639
640     leftSet = CanonicalDecNodes2D(root->left, b, t);
641     rightSet = CanonicalDecNodes2D(root->right, b, t);
642
643     UnionNodeSets2D(leftSet, rightSet);
644
645     // If the input range entirely encompasses the range
646     // of this node, then return me in addition to my decendants
          .
647
648     if (nodeWithinRange2D(root, b, t)) {
649       mySet = UnitNodeSet2D(root);
650       UnionNodeSets2D(leftSet, mySet);
651     }
652
653     return(leftSet);
654 }
655
656 // Inserts the specified rectangle into the 2D segment tree
657 // O(lg^{2}n + D)
658
659 void InsertRect(SegTreeNode2D * T, Num l, Num r, Num b, Num t)
660 {
661     if (T == NULL) return;
662
663     if (!checkLRBTBounds(l,r,b,t)) return;
664
665     if (!nodeOverlapsRange2D(T, b, t)) return;
666
667     InsertRange(T->projtree, l, r);
668     // If this is a canonical node
669     if (nodeWithinRange2D(T, b, t)) {
670       InsertRange(T->perptree, l, r);
671       return;
672     }
673
674     InsertRect(T->left, l, r, b, t);
675     InsertRect(T->right, l, r, b, t);
676 }
677
678 // Inserts the specificed rectangle, striped in the X
      direction
679 // according to XTree, into the 2D segment tree
```

```
680 // O(nlgn + D)
681
682 void InsertStripedRectX(SegTreeNode2D * XYTree, SegTreeNode *
        XTree,
683 Num l, Num r)
684 {
685   if (XYTree == NULL) return;
686   if (XTree == NULL) return;
687   if (!checkLRBounds(l, r)) return;
688
689   if (!nodeOverlapsRange2D(XYTree, XTree->l, XTree->r)) return
          ;
690
691   // If I'm a cannonical ancestor
692   if (!nodeWithinRange2D(XYTree, XTree->l, XTree->r)) {
693     InsertStripedRectX(XYTree->left, XTree, l, r);
694     InsertStripedRectX(XYTree->right, XTree, l, r);
695     return;
696   }
697
698   if (XTree->data) InsertRange(XYTree->perptree, l, r);
699   if (XTree->subtreedata) InsertRange(XYTree->projtree, l, r);
700
701   InsertStripedRectX(XYTree, XTree->left, l, r);
702   InsertStripedRectX(XYTree, XTree->right, l, r);
703 }
704
705 // Inserts the specificed rectangle, striped in the Y
        direction
706 // according to YTree, into the 2D segment tree
707 // O(nlgn + D)
708
709 void InsertStripedRectY(SegTreeNode2D * XYTree, SegTreeNode *
        YTree,
710 Num b, Num t)
711 {
712   if (XYTree == NULL) return;
713   if (YTree == NULL) return;
714   if (!checkBTBounds(b, t)) return;
715
716   if (!nodeOverlapsRange2D(XYTree, b, t)) return;
717
718   UnionTrees(XYTree->projtree, YTree);
719   // If this is a canonical node
720   if (nodeWithinRange2D(XYTree, b, t)) {
```

```
721       UnionTrees(XYTree->perptree, YTree);
722       return;
723     }
724
725     InsertStripedRectY(XYTree->left, YTree, b, t);
726     InsertStripedRectY(XYTree->right, YTree, b, t);
727   }
728
729   // Insterts the specificed rectangle, striped in the sDir
        direction
730   // according to Tree, into the 2D segment tree
731   // O(nlgn + D)
732
733   void InsertStripedRect(SegTreeNode2D * XYTree, SegTreeNode *
        Tree,
734   Num l, Num r, Num b, Num t, Axis sDir)
735   {
736     if (!checkLRBTBounds(l, r, b, t)) return;
737
738     if (sDir == XAxis) {
739       if(QueryRange(Tree, Tree->l, b-1) ||
740         QueryRange(Tree, t+1, Tree->r))
741         genError(ERROR, "InsertStripedRect",
742           "data outside rect in InsertRectX");
743       InsertStripedRectX(XYTree, Tree, l, r);
744     }
745     if (sDir == YAxis) {
746       if(QueryRange(Tree, Tree->l, l-1) ||
747         QueryRange(Tree, r+1, Tree->r))
748         genError(ERROR, "InsertStripedRect",
749           "data outside rect in InsertRectY");
750       InsertStripedRectY(XYTree, Tree, b, t);
751     }
752   }
753
754   // Return the projection of the data in XYTree within the
        specified
755   // rectangle onto the X Axis
756   // O(nlgn)
757
758   SegTreeNode * ProjectRectX(SegTreeNode2D * XYTree, Num l, Num
        r, Num b, Num t)
759   {
760     SegTreeNode * XTree;
761
```

```
762     if (XYTree == NULL) return NULL;
763
764     if(!nodeOverlapsRange2D(XYTree, b, t)) return NULL;
765
766     XTree = newSegTree(XYTree->b, XYTree->t);
767     XTree->data = XTree->subtreedata = FALSE;
768
769     // If this is a canonical node, ancestor, or decendant with
            data
770
771     if (QueryRange(XYTree->perptree, l, r)) {
772     //  Insert the intersection of b,t and XYTree->b,XYTree->t
773       InsertRange(XTree, b, t);
774       return XTree;
775     }
776
777     XTree->left = ProjectRectX(XYTree->left, l, r, b, t);
778     XTree->right = ProjectRectX(XYTree->right, l, r, b, t);
779     maintainNode(XTree);
780
781     return XTree;
782   }
783
784   // Return the projection of the data in XYTree within the
        specified
785   // rectangle onto the Y Axis
786   // O(nlgn)
787
788   SegTreeNode * ProjectRectY(SegTreeNode2D * XYTree, Num l, Num
        r, Num b, Num t)
789   {
790     SegTreeNode * YTree;
791     SegTreeNode * leftTree;
792     SegTreeNode * rightTree;
793
794     if (!nodeOverlapsRange2D(XYTree, b, t)) return NULL;
795
796     YTree = newSegTree(XYTree->perptree->l, XYTree->perptree->r)
            ;
797     YTree->data = YTree->subtreedata = FALSE;
798
799     // If this is a canonical node
800     if (nodeWithinRange2D(XYTree, b, t)){
801       UnionTrees(YTree, XYTree->projtree);
802       TruncateTree(YTree, l, r);
```

```
803       return YTree;
804     }
805
806     // This is an ancestor
807     leftTree = ProjectRectY(XYTree->left, l, r, b, t);
808     rightTree = ProjectRectY(XYTree->right, l, r, b, t);
809
810     UnionTrees(YTree, XYTree->perptree);
811     UnionTrees(YTree, leftTree);
812     UnionTrees(YTree, rightTree);
813     TruncateTree(YTree, l, r);
814
815     free(leftTree);
816     free(rightTree);
817
818     return YTree;
819
820 }
821
822
823 // Return the projection of the data in XYTree within the
        specified
824 // rectangle onto the pDir Axis
825 // O(nlgn)
826
827 SegTreeNode * ProjectRect(SegTreeNode2D * XYTree, Num l, Num r
        ,
828 Num b, Num t, Axis pDir)
829 {
830     if (!checkLRBTBounds(l, r, b, t)) return NULL;
831     if (pDir == XAxis) return ProjectRectX(XYTree, l, r, b, t);
832     if (pDir == YAxis) return ProjectRectY(XYTree, l, r, b, t);
833 }
834
835 // Return TRUE if any data in T falls within the given
        rectangle.
836 // O(nlgn)
837
838 Boolean QueryRect(SegTreeNode2D * T, Num l, Num r, Num b, Num
        t)
839 {
840     SegTreeNode * YTree;
841
842     if (T == NULL) return FALSE;
843     if (!checkLRBTBounds(l, r, b, t)) return FALSE;
```

```
844
845   if (!nodeOverlapsRange2D(T, b, t)) return FALSE;
846
847   if (QueryRange(T->perptree, l, r)) return TRUE;
848
849   return (QueryRect(T->left, l, r, b, t) ||
850     QueryRect(T->right, l, r, b, t));
851 }
852
853
854 // Remove all data stored in T within the indicated rectangle
855 // O(nlgn + D)
856
857 void ClearRect(SegTreeNode2D * T, Num l, Num r, Num b, Num t)
858 {
859   if (T == NULL) return;
860   if (!checkLRBTBounds(l, r, b, t)) return;
861
862   if (!nodeOverlapsRange2D(T, b, t)) return;
863
864   // If I'm a cannonical ancestor
865   if (!nodeWithinRange2D(T, b, t)) {
866     if ((!T->left) || (!T->right)) {
867       genError(ERROR, "ClearRect",
868         "Canonical Ancestor has NULL Child");
869       return;
870     }
871
872     // Push my data onto my children
873     UnionTrees(T->left->perptree, T->perptree);
874     UnionTrees(T->left->projtree, T->perptree);
875     UnionTrees(T->right->perptree, T->perptree);
876     UnionTrees(T->right->projtree, T->perptree);
877
878     // Clear my data
879     ClearChildrenSubtrees(T->perptree);
880     T->perptree->data = T->perptree->subtreedata = FALSE;
881   }
882
883   // If I'm a cannonical node or decendant
884   else {
885     ClearRange(T->perptree, l, r);
886     ClearRange(T->projtree, l, r);
887   }
888
```

```
889     if (T->left) ClearRect(T->left, l, r, b, t);
890     if (T->right) ClearRect(T->right, l, r, b, t);
891
892     // If I'm a cannonical ancestor maintain my projtree
893
894     if (!nodeWithinRange2D(T, b, t)) maintainNode2D(T);
895 }
```

Listing 8.2: segtree.c

Source code for :subface.c

```
1  // subface.c
       David Wagner
2  // Operations on SubFaces
3
4  #include "include.h"
5
6  #include "subface.h"
7  #include "space.h"
8  #include "io.h"
9
10 //    Coordinate system with dimension and sign
11 //
12 //        Y
13 //        |    Z
14 //        |   /
15 //        |  /
16 //        |/______ X
17
18 // Return a new SubFace with the given bounds and direction.
19
20 SubFace * newSubFace(Num bound[3][2], Axis axis, Sign sign,
       FaceType facetype)
21 {
22   int i,j;
23   SubFace * newFace = (SubFace *) malloc(sizeof(SubFace));
24
25   for (i=XAxis;i<=ZAxis;i++){
26     newFace->bound[i][POS] = bound[i][POS];
27     newFace->bound[i][NEG] = bound[i][NEG];
28     newFace->lastsweep[i][POS] = -1;
29     newFace->lastsweep[i][NEG] = -1;
30   }
31
32   newFace->sign = sign;
33   newFace->axis = axis;
34   newFace->facetype = facetype;
35   newFace->coord = newFace->bound[axis][POS];
36
37   generateFaceDirs(newFace, axis, sign);
38
39   return newFace;
40 }
```

```
41
42
43  // Returns a new SubFaceSet node, specially marked to be
44  // the head node of the SubFaceSet
45
46  SubFaceSet * newSubFaceSetHead()
47  {
48    SubFaceSet * newSubFaceSet = (SubFaceSet *) malloc(sizeof(
        SubFaceSet));
49
50    newSubFaceSet->face = NULL;
51    newSubFaceSet->count = 0;
52    newSubFaceSet->next = newSubFaceSet;
53    newSubFaceSet->prev = newSubFaceSet;
54  }
55
56  // Checks if the SubFaceSet node is the head of the SubFaceSet
57
58  Boolean isSubFaceSetHead(SubFaceSet * subfaceset)
59  {
60    return (subfaceset->face == NULL);
61  }
62
63  // Add a SubFace to a SubFaceSet
64
65  void insertSubFace(SubFaceSet * head, SubFace * subface)
66  {
67    SubFaceSet * newSubFaceSet = (SubFaceSet *) malloc(sizeof(
        SubFaceSet));
68
69    newSubFaceSet->face = subface;
70    newSubFaceSet->next = head->next;
71    newSubFaceSet->prev = head;
72    head->next->prev = newSubFaceSet;
73    head->next = newSubFaceSet;
74    head->count++;
75
76  }
77
78  // Remove the SubFace indicated by the given SubFaceSet node
      from
79  // its SubFaceSet.  Return the subface which was removed.
80
81  SubFace * removeSubFace(SubFaceSet * head, SubFaceSet *
      subface)
```

```
82  {
83    SubFace * result;
84
85    subface->prev->next = subface->next;
86    subface->next->prev = subface->prev;
87
88    result = subface->face;
89
90    head->count--;
91
92    free(subface);
93
94    return result;
95  }
96
97  // Destructively union two subfacesets together
98  // Free the old head, and return the new head
99
100 SubFaceSet * unionSubFaceSets(SubFaceSet * set1, SubFaceSet *
        set2)
101 {
102   set2->prev->next = set1;
103   set1->prev->next = set2->next;
104
105   set2->next->prev = set1->prev;
106   set2->next = set2;
107   set2->prev->next->prev = set2->prev;
108
109   set1->count += set2->count;
110
111   free(set2);
112
113   return(set1);
114 }
115
116
117 // Take a single face and split it along the plane
118 // defined by the given axis and given coordinate
119 // Return the resulting face on either the positive
120 // or the negative side, as indicated by the last argument
121
122 SubFace * splitFace(SubFace * subface, Axis axis, Num coord,
        Sign side)
123 {
124   SubFace * newFace;
```

```
125   Num newBound[3][2];
126   int i,j;
127
128   if(!planeIntersectsSubFace(subface, axis, coord) ||
129     !planeIntersectsSubFace(subface, axis, coord+1))
130     genError(ERROR, "splitFace", "Splitting face on non-
          intersecting plane\n");
131
132   for(i=XAxis;i<=ZAxis;i++)
133     for(j=0; j<=1; j++)
134       newBound[i][j]=subface->bound[i][j];
135   newBound[axis][1-side]=coord+side;
136   newFace = newSubFace(newBound, subface->axis, subface->sign,
        subface->facetype);
137
138   return(newFace);
139 }
140
141 SubFace * splitFaceOnRealPlane(SubFace * subface, Axis axis,
     Num coord, Sign side)
142 {
143   SubFace * newFace;
144   Num newBound[3][2];
145   int i,j;
146
147   if(!realPlaneIntersectsSubFace(subface, axis, coord))
148     genError(ERROR, "splitFaceOnRealPlane", "Splitting face on
          non-intersecting plane\n");
149
150   for(i=XAxis;i<=ZAxis;i++)
151     for(j=0; j<=1; j++)
152       newBound[i][j]=subface->bound[i][j];
153   newBound[axis][1-side]=coord+side-1;
154   newFace = newSubFace(newBound, subface->axis, subface->sign,
        subface->facetype);
155
156   return(newFace);
157 }
158
159
160 // Return all faces on one side of the given plane
161 // Split any faces crossing the plane and return the
162 // half on the correct side.
163
```

```
164 SubFaceSet * splitAllFacesInSubFaceSet(SubFaceSet * head, Axis
        axis, Num coord,
165     Sign side)
166 {
167   SubFaceSet * result = newSubFaceSetHead();
168   SubFaceSet * currface;
169
170   // For each face
171   for(currface = head->next; currface!=head; currface =
        currface->next) {
172
173     if (subFaceInRealPlane(currface->face, axis, coord)){
174       // If the subface lies in the plane, discard it
175       // do nothing
176     } else if(realPlaneIntersectsSubFace(currface->face, axis,
          coord))
177       // If the subface is intersected by the plane
178       // Split the subface and put the correct half
179       // into the set
180       insertSubFace(result, splitFaceOnRealPlane(currface->
            face,
181                 axis, coord, side));
182      else if (subFaceOnRealSide(currface->face, axis, coord,
          side))
183       // if the subface is on the correct side
184       // put the subface into the the set
185       insertSubFace(result,currface->face);
186
187   }
188
189   return result;
190
191 }
192
193
194 Space * splitAllFacesInSpace(Space * space, Axis axis, Num
      coord, Sign side)
195 {
196   Space * result;
197   Num newBound[3][2];
198
199   int i,j;
200
201   for(i=XAxis; i<=ZAxis; i++)
202     for(j=0;j<=1;j++){
```

```
203         newBound[i][j]=space->bound[i][j];
204       }
205
206     newBound[axis][1-side]=coord + side-1;
207
208     result = newSpace(newBound);
209
210     for(i=XAxis; i<=ZAxis; i++)
211       for(j=0;j<=1;j++){
212         result->faces[i][j]=
213           splitAllFacesInSubFaceSet(space->faces[i][j],
214           axis, coord, side);
215       }
216
217     countAllFacesInSpace(result);
218
219     return result;
220 }
221
222
223 // Return TRUE if the subface falls along the real plane
224
225 Boolean subFaceInRealPlane(SubFace * subface, Axis axis, Num
        coord)
226 {
227     checkFaceBoundsOnAxis(subface, axis);
228     if (subface->axis != axis) return FALSE;
229     return (getRealFaceCoord(subface) == coord);
230 }
231
232
233 // Return TRUE if the subface is completely on the indicated
        side of the plane
234
235 Boolean subFaceOnRealSide(SubFace * subface, Axis axis, Num
        coord, Sign side)
236 {
237     checkFaceBoundsOnAxis(subface, axis);
238
239     if(side == POS) return (subface->bound[axis][NEG] >= coord);
240 //        && subface->bound[axis][POS]+1 > coord
241     if(side == NEG)
242       return (subface->bound[axis][POS]+1 <= coord);
243 //      && subface->bound[axis][NEG] < coord
244 }
```

```
245
246
247 // Return the number of faces in the faceset
248
249 int countFaces(SubFaceSet * head)
250 {
251   int i=0;
252   SubFaceSet * currface;
253
254 //   printf("Counting Faces\n");
255 //   for(currface = head->next; currface!=head; currface =
        currface->next)
256 //     i++;
257 //
258 //   if(i!=head->count)
259 //     printf("Error: Face counts do not match\n");
260 //   return i;
261
262   return head->count;
263 }
264
265 // Moved to space.c
266
267 //int countAllFacesInSpace(Space * space)
268 //{
269 //   int i,j;
270 //   int accum;
271 //
272 //   for(i=XAxis; i<=ZAxis; i++)
273 //     for(j=0;j<=1;j++){
274 //       space->facecount[i][j]=countFaces(space->faces[i][j]);
275 //       accum+=space->facecount[i][j];
276 //     }
277 //   return accum;
278 //}
279
280
281 Boolean planeOfSubFaceIntersectsSubFace(SubFace * plane,
      SubFace * subface)
282 {
283   return planeIntersectsSubFace(subface, plane->axis, plane->
        coord);
284 }
285
```

```
286 Boolean realPlaneIntersectsSubFace(SubFace * subface, Axis
        axis, Num coord)
287 {
288   checkFaceBoundsOnAxis(subface, axis);
289         return (subface->bound[axis][NEG] < coord &&
290                 coord <= subface->bound[axis][POS]);
291 }
292
293
294 Boolean planeIntersectsSubFace(SubFace * subface, Axis axis,
        Num coord)
295 {
296   checkFaceBoundsOnAxis(subface, axis);
297         return (subface->bound[axis][NEG] <= coord &&
298                 coord <= subface->bound[axis][POS]);
299 }
300
301
302 Boolean planeSplitsSubFace(SubFace * subface, Axis axis, Num
        coord)
303 {
304   if(!planeIntersectsSubFace(subface, axis, coord)) return
        FALSE;
305   if(!planeIntersectsSubFace(subface, axis, coord+1)) return
        FALSE;
306   return TRUE;
307 }
308
309
310 // Add all six faces bounding the represented region to
311 // the subfaceset
312
313 // depricated
314
315 //void addSubFaceCubeToSubFaceSet(Num bound[3][2], SubFaceSet
        * head)
316 //{
317 //   SubFace * tempface;
318 //   Num tempbound[3][2];
319 //   int i, j, k, l;
320 //
321 //   for(i=XAxis; i<=ZAxis; i++) {
322 //     for(j=0; j<=1; j++) {
323 //       for(k=XAxis; k<=ZAxis; k++)
324 //         for(l=0; l<=1; l++)
```

```
325 //              tempbound[k][l]=bound[k][l];
326 //          for(l=0; l<=1; l++)
327 //            tempbound[i][l] = bound[i][j];
328 //          tempface = newSubFace(tempbound, i, j);
329 //          insertSubFace(head, tempface);
330 //      }
331 //  }
332
333
334
335 //  SubFace * xfacepos, * xfaceneg,
336 //      * yfacepos, * yfaceneg,
337 //      * zfacepos, * zfaceneg;
338 //
339 //  xfacepos = newSubFace({
340 //          {bound[XAxis][POS], bound[XAxis][POS]},
341 //          {bound[YAxis][0], bound[YAxis][1]},
342 //          {bound[ZAxis][0], bound[ZAxis][1]}},
343 //          XAxis, POS);
344 //
345 //  xfaceneg = newSubFace({
346 //          {bound[XAxis][NEG], bound[XAxis][NEG]},
347 //          {bound[YAxis][0], bound[YAxis][1]},
348 //          {bound[ZAxis][0], bound[ZAxis][1]}},
349 //          XAxis, NEG);
350 //
351 //  yfacepos = newSubFace({
352 //          {bound[YAxis][0], bound[YAxis][1]},
353 //          {bound[XAxis][POS], bound[XAxis][POS]},
354 //          {bound[ZAxis][0], bound[ZAxis][1]}},
355 //          YAxis, POS);
356 //
357 //  yfaceneg = newSubFace({
358 //          {bound[YAxis][0], bound[YAxis][1]},
359 //          {bound[XAxis][NEG], bound[XAxis][NEG]},
360 //          {bound[ZAxis][0], bound[ZAxis][1]}},
361 //          YAxis, NEG);
362 //
363 //  zfacepos = newSubFace({
364 //          {bound[YAxis][0], bound[YAxis][1]},
365 //          {bound[ZAxis][0], bound[ZAxis][1]}},
366 //          {bound[XAxis][POS], bound[XAxis][POS]},
367 //          ZAxis, POS);
368 //
369 //  zfaceneg = newSubFace({
```

```
370 //          {bound[YAxis][0], bound[YAxis][1]},
371 //          {bound[ZAxis][0], bound[ZAxis][1]}},
372 //          {bound[XAxis][NEG], bound[XAxis][NEG]},
373 //          ZAxis, NEG);
374
375 //}
376
377
378 // Return a random subface from the SubFaceSet of size count
379
380 SubFace * GetRandomFaceWithCount (SubFaceSet * head, int count
        )
381 {
382   int i;
383   int randomNum = random() % count;
384   SubFaceSet * currface = head->next;
385
386   for(i=0; i<randomNum; i++) currface = currface->next;
387
388   return currface->face;
389 }
390
391 SubFace * GetRandomFaceInSubFaceSet (SubFaceSet * head)
392 {
393   int count = countFaces(head);
394   return (GetRandomFaceWithCount(head, count));
395 }
396
397 SubFace * GetRandomFaceInSpace(Space * space)
398 {
399   int count = countAllFacesInSpace(space);
400   int randomNum = random() % count;
401   int i,j;
402
403   for(i=XAxis; i<=ZAxis; i++) for(j=0; j<=1; j++){
404     if (randomNum < countFaces(space->faces[i][j]))
405       return GetRandomFaceInSubFaceSet(space->faces[i][j]);
406     randomNum -= countFaces(space->faces[i][j]);
407   }
408
409 }
410
411 //SubFace * GetRandomFace (SubFaceSet * head)
412 //{
413 //   int count = head->count;
```

```
414 //  return (GetRandomFaceWithCount(head, count));
415 //}
416
417 // Returns true if the coord is between but not on the
        boundaries of the face
418
419 Boolean RealCoordDividesFaceBoundaries(SubFace * face, Axis
        axis, Num coord)
420 {
421   if (axis == face->axis) return FALSE;
422   return (face->bound[axis][NEG] < coord && coord < face->
          bound[axis][POS]+1);
423 }
424
425 // Returns true if the coord is on the boundary of a face
426
427 Boolean RealCoordOnFaceBoundary(SubFace * face, Axis axis, Num
         coord, Sign sign)
428 {
429   if (axis == face->axis)
430     return (coord == face->bound[axis][face->sign] + face->
            sign);
431   return (face->bound[axis][sign]+sign == coord);
432 }
433
434 // Returns true if the coord is between or on the boundaries
        of a face
435
436 Boolean RealCoordTouchesFace(SubFace * face, Axis axis, Num
        coord)
437 {
438   if (axis == face->axis)
439     return(coord == face->bound[axis][face->sign] + face->sign
            );
440   return (face->bound[axis][NEG] < coord &&
441     coord < face->bound[axis][POS]+1);
442 }
443
444
445 // Returns the POS or NEG real boundary of the face
446
447 Num GetRealFaceBoundary(SubFace * face, Axis axis, Sign sign)
448 {
449   if (axis == face->axis)
450     return (face->bound[axis][face->sign] + face->sign);
```

```
451     return(face->bound[axis][sign]+sign);
452 }
453
454
455 // Return true if the two subfaces share any overlap range
456 // along the given axis
457
458 Boolean SubFacesOverlapAlongAxis(SubFace * face1, SubFace *
        face2, Axis axis)
459 {
460     return (face1->bound[axis][NEG]<=face2->bound[axis][POS] &&
461             face2->bound[axis][NEG]<=face1->bound[axis][POS]);
462 }
463
464
465 // Return true if face1 intersects face2 between its real
        boundaries
466
467 Boolean FaceDividesFace(SubFace * face1, SubFace * face2)
468 {
469     Axis altAxis = XAxis + YAxis + ZAxis - face1->axis - face2->
          axis;
470
471     if (SubFacesCoplanar(face1, face2)) return FALSE;
472
473     if(!RealCoordDividesFaceBoundaries(face2, face1->axis,
474       GetRealFaceBoundary(face1, face1->axis, POS)))
475         return FALSE;
476     if(!RealCoordTouchesFace(face1, face2->axis,
477       GetRealFaceBoundary(face2, face2->axis, POS)))
478         return FALSE;
479     if(!SubFacesOverlapAlongAxis(face1, face2, altAxis))
480       return FALSE;
481 //    printf("Faces split each other\n");
482 //    printFace(face1);
483 //    printFace(face2);
484     return TRUE;
485 }
486
487
488 // Return true if the coordinates of the two subfaces are
489 // coplanar, and differ by one along their axis
490
491 Boolean SubFacesFaceEachOther(SubFace * face1, SubFace * face2
        )
```

```
492 {
493   Axis axis = face1->axis;
494
495   if(face1->axis != face2->axis) return FALSE;
496   if(face1->sign == face2->sign) return FALSE;
497
498   return (face1->bound[axis][face1->sign] + face1->sign ==
499     face2->bound[axis][face2->sign] + face2->sign);
500 }
501
502
503 // Return true if the two subfaces face each other, and
504 // share the same coordinates in the other two dimensions
505
506 Boolean SubFacesFaceAndAreFlush(SubFace * face1, SubFace *
      face2)
507 {
508   Axis axis = face1->axis;
509   int i,j;
510
511   if (!SubFacesFaceEachOther(face1,face2)) return FALSE;
512   for (i=(axis+1)%3; i!=axis; i=(i+1)%3)
513     for (j=0; j<=1; j++)
514       if (face1->bound[i][j] != face2->bound[i][j]){
515         return FALSE;
516       }
517   return TRUE;
518 }
519
520 // Return true if the two subfaces face each other, and
521 // share the same coordinates in the other two dimensions
522 // And one or the other is the start point or finish point
523
524 Boolean SubFacesFaceAndAreFlushWithPoint(SubFace * face1,
      SubFace * face2)
525 {
526   if (face1->facetype == START || face1->facetype == FINISH ||
527       face2->facetype == START || face2->facetype == FINISH)
528     if(SubFacesFaceAndAreFlush(face1, face2)){
529 //       printf("Subface flush with start or finish point\n");
530       return TRUE;
531     }
532   return FALSE;
533 }
534
```

```
535 // Return true if the two subfaces face each other, and
536 // overlap along both of of the other axes
537 // WARNING: This returns true if the faces are flush!
538
539 Boolean SubFacesFaceAndOverlapEachOther(SubFace * face1,
        SubFace * face2)
540 {
541   Axis axis = face1->axis;
542   int i;
543
544   if (!SubFacesFaceEachOther(face1,face2)) return FALSE;
545   for (i=(axis+1)%3; i!=axis; i=(i+1)%3){
546     if (face1->bound[i][POS] < face2->bound[i][NEG]) {
547 //      printf("Subfaces do not overlap on axis %d\n",i);
548       return FALSE;}
549     if (face2->bound[i][POS] < face1->bound[i][NEG]) {
550 //      printf("Subfaces do not overlap on axis %d\n",i);
551       return FALSE;}
552   }
553 //  printf("Subfaces face and overlap\n");
554 //  printFace(face1);
555 //  printFace(face2);
556   return TRUE;
557 }
558
559 // Return true if the two subfaces are coplanar, and
560 // overlap along both of of the other axes
561
562 Boolean SubFacesCoplanarAndOverlapEachOther(SubFace * face1,
        SubFace * face2)
563 {
564   Axis axis = face1->axis;
565   int i;
566
567   if (!SubFacesCoplanar(face1,face2)) return FALSE;
568   for (i=(axis+1)%3; i!=axis; i=(i+1)%3){
569     if (face1->bound[i][POS] < face2->bound[i][NEG]) {
570 //      printf("Subfaces do not overlap on axis %d\n",i);
571       return FALSE;}
572     if (face2->bound[i][POS] < face1->bound[i][NEG]) {
573 //      printf("Subfaces do not overlap on axis %d\n",i);
574       return FALSE;}
575   }
576 //  printf("Subfaces coplanar and overlap\n");
577 //  printFace(face1);
```

```
578 //   printFace(face2);
579   return TRUE;
580 }
581
582
583 //Boolean SubFacesFormCorner(SubFace * face1, SubFace * face2)
584 //{
585 //
586 //}
587
588
589 Boolean SubFacesFormIllegalCorner(SubFace * face1, SubFace *
        face2)
590 {
591   return (FaceDividesFace(face1, face2) ||
592     FaceDividesFace(face2, face1));
593 }
594
595
596 //Boolean SubFacesFormLegalCorner(SubFace * face1, SubFace *
        face2)
597 //{
598 //
599 //}
600
601
602 Boolean SubFacesCoplanar(SubFace * face1, SubFace * face2)
603 {
604   return(face1->axis == face2->axis &&
605     face1->coord == face2->coord &&
606     face1->sign == face2->sign);
607 }
608
609 Boolean SubFaceFacesSpaceWall(SubFace * face, Space * space)
610 {
611   Axis axis = face->axis;
612   Sign sign = face->sign;
613
614   return (face->bound[axis][sign] == space->bound[axis][sign])
          ;
615 }
616
617 Boolean UpdateInsideSpaceWall(SubFace * face, Space * space)
618 {
619   Axis axis = face->axis;
```

```
620   Sign sign = face->sign;
621         Num l, r, b, t;
622
623         l = face->bound[generateLRAxis(axis)][generateLeftSign
              (sign)];
624         r = face->bound[generateLRAxis(axis)][
              generateRightSign(sign)];
625         b = face->bound[generateTBAxis(axis)][
              generateBottomSign(sign)];
626         t = face->bound[generateTBAxis(axis)][generateTopSign(
              sign)];
627
628   if (PRINT_ADDING_CUBE){
629     printf("Adding cube to face on axis %d, sign %d, ",axis,
          sign);
630     printf("Coords L: %d, R: %d, B: %d, T: %d\n",l,r,b,t);
631   }
632
633         if (QueryRect(space->inside[axis][sign], l, r, b, t))
              {
634     if (PRINT_ADDING_CUBE) printTree2D(space->inside[axis][
          sign]);
635     return FALSE;
636   }
637         InsertRect(space->inside[axis][sign], l, r, b, t);
638   return TRUE;
639
640 }
641
642 Boolean SubFacesIllegal(SubFace * face1, SubFace * face2)
643 {
644   if (SubFacesFaceAndAreFlushWithPoint(face1, face2)) return
        TRUE;
645   if (SubFacesFaceAndAreFlush(face1, face2)) return FALSE;
646   if (SubFacesFaceAndOverlapEachOther(face1, face2)) return
        TRUE;
647   if (SubFacesCoplanarAndOverlapEachOther(face1, face2))
        return TRUE;
648   return (SubFacesFormIllegalCorner(face1, face2));
649 }
650
651 Boolean SubFaceLegalInSubFaceSet(SubFace * face, SubFaceSet *
      head)
652 {
653   SubFaceSet * currface;
```

```
654
655   for(currface = head->next; currface!=head; currface =
          currface->next){
656     if (SubFacesIllegal(face, currface->face)) {
657 //        genError(NOTICE, "SubFaceLegalInSubFaceSet",
658 //          "Faces conflict");
659 //        printFace(face);
660 //        printFace(currface->face);
661         return FALSE;
662     }
663   }
664
665   return TRUE;
666 }
667
668 Boolean SubFaceLegalInSpace(SubFace * face, Space * space)
669 {
670   int i, j;
671
672   for (i=XAxis; i<=ZAxis; i++)
673     for (j=0; j<=1; j++)
674       if(!SubFaceLegalInSubFaceSet(face, space->faces[i][j]))
675         return FALSE;
676   return TRUE;
677 }
678
679 Boolean SubFaceFlushInSubFaceSet(SubFace * face, SubFaceSet *
      head)
680 {
681   SubFaceSet * currface;
682
683   for(currface = head->next; currface!=head; currface =
          currface->next){
684     if (SubFacesFaceAndAreFlush(face, currface->face)) {
685 //        printf("Subfaces Flush\n");
686 //        printFace(face);
687 //        printFace(currface->face);
688         return TRUE;
689     }
690   }
691   return FALSE;
692 }
693
694 Boolean SubFaceFlushInSpace(SubFace * face, Space * space)
695 {
```

```
696   Axis axis = face->axis;
697   Sign sign = 1-face->sign;
698
699   return(SubFaceFlushInSubFaceSet(face, space->faces[axis][
          sign]));
700 }
701
702
703 SubFace * RemoveFlushSubFaceInSubFaceSet(SubFace * face,
        SubFaceSet * head)
704 {
705   SubFaceSet * currface;
706   SubFace * result;
707
708   for(currface = head->next; currface!=head; currface =
          currface->next){
709     if (SubFacesFaceAndAreFlush(face, currface->face)) {
710       if (PRINT_INSERTED_SUBFACE){
711         printf("Removing Flush Face\n");
712         printFace(currface->face);
713       }
714       result = removeSubFace(head, currface);
715       return result;
716     }
717   }
718   return NULL;
719 }
720
721
722 SubFace * RemoveFlushSubFaceInSpace(SubFace * face, Space *
        space)
723 {
724   Axis axis = face->axis;
725   Sign sign = 1-face->sign;
726
727   return (RemoveFlushSubFaceInSubFaceSet(face, space->faces[
          axis][sign]));
728 }
729
730
731
732 Boolean TestAndInsertSubFaceIntoSpace(SubFace * face, Space *
        space)
733 {
734   Axis axis = face->axis;
```

```
735   Sign sign = face->sign;
736   SubFace * removedFace;
737
738 //  printf("Attempting to insert face ");
739 //  printBound(face->bound);
740 //  printFace(face);
741 //  printf("\n");
742
743   if (!SubFaceLegalInSpace(face, space)) return FALSE;
744   if (SubFaceFlushInSpace(face, space)){
745     removedFace = RemoveFlushSubFaceInSpace(face, space);
746     free(removedFace);
747     return TRUE;
748   }
749
750   if (SubFaceFacesSpaceWall(face, space)){
751     if (!UpdateInsideSpaceWall(face, space))
752 //      genError(ERROR, "TestAndInsertSubFaceIntoSpace",
753 //      "Tried to insert into occupied wall space\n");
754     if (PRINT_INSERTED_SUBFACE){
755       printf("Subface faces wall, not inserted\n");
756       printFace(face);
757     }
758     return TRUE;
759   }
760
761   insertSubFace(space->faces[axis][sign], face);
762   space->facecount[axis][sign]++;
763   if (PRINT_INSERTED_SUBFACE){
764     printf("Inserted subface, type = %d\n",face->facetype);
765     printFace(face);
766   }
767   return TRUE;
768 }
769
770 Boolean SubFaceCubeBoundsReversed(Num bound[3][2])
771 {
772   int i;
773
774   for(i=XAxis; i<=ZAxis; i++)
775     if (bound[i][POS] < bound[i][NEG]) {
776       genError(NOTICE, "SubFaceCubeBoundsReversed",
777         "Min greater than Max");
778       return TRUE;
779     }
```

```
780
781   return FALSE;
782
783 }
784
785 Boolean SubFaceCubeBeyondSpaceBounds(Num bound[3][2], Space *
      space)
786 {
787   int i;
788
789   for(i=XAxis; i<=ZAxis; i++)
790     if (bound[i][POS] > space->bound[i][POS] ||
791         bound[i][NEG] < space->bound[i][NEG]){
792 //       genError(NOTICE, "SubFaceCubeBeyondSpaceBounds",
793 //         "Obstacle beyond bounds of Space");
794       return TRUE;
795     }
796
797   return FALSE;
798 }
799
800 Boolean PointInsideSubFaceCube(Point * p, Num bound[3][2])
801 {
802   if (p->x < bound[XAxis][NEG]) return FALSE;
803   if (p->x > bound[XAxis][POS]) return FALSE;
804   if (p->y < bound[YAxis][NEG]) return FALSE;
805   if (p->y > bound[YAxis][POS]) return FALSE;
806   if (p->z < bound[ZAxis][NEG]) return FALSE;
807   if (p->z > bound[ZAxis][POS]) return FALSE;
808   return TRUE;
809 }
810
811 Boolean StartOrFinishPointInsideSubFaceCube(Num bound[3][2],
      Space * space)
812 {
813   if (space->start)
814   if (PointInsideSubFaceCube(space->start, bound)) {
815 //     genError(NOTICE, "StartOrFinishPointInsideSubFaceCube",
816 //       "Start Point inside Obstacle");
817     return TRUE;
818   }
819   if (space->finish)
820   if (PointInsideSubFaceCube(space->finish, bound)) {
821 //     genError(NOTICE, "StartOrFinishPointInsideSubFaceCube",
822 //       "Finish Point inside Obstacle");
```

```
823       return TRUE;
824     }
825     return FALSE;
826   }
827
828
829   Boolean FaceInsideCube (Num bound[3][2], SubFace * face)
830   {
831     Axis axis;
832
833     for(axis=XAxis; axis<=ZAxis; axis++){
834       if (face->bound[axis][POS] < bound[axis][NEG]) return
              FALSE;
835       if (face->bound[axis][POS] > bound[axis][POS]) return
              FALSE;
836     }
837
838     return TRUE;
839   }
840
841   Boolean AnyFaceFromSubFaceSetInsideCube (Num bound[3][2],
        SubFaceSet * head)
842   {
843     SubFaceSet * currface;
844
845     for (currface = head->next; currface!=head; currface=
          currface->next){
846       if (FaceInsideCube(bound, currface->face))
847         return TRUE;
848     }
849
850     return FALSE;
851   }
852
853   Boolean AnyFaceInsideSubFaceCube (Num bound[3][2], Space *
        space)
854   {
855     int i, j;
856
857     for (i=XAxis; i<=ZAxis; i++) for(j=0; j<=1; j++) {
858       if (AnyFaceFromSubFaceSetInsideCube(bound, space->faces[i
              ][j]))
859         return TRUE;
860     }
861
```

```
862     return FALSE;
863 }
864
865
866 Boolean SubFaceCubeLegalInSpace(Num bound[3][2], Space * space
        )
867 {
868   SubFace * tempface;
869   Num tempbound[3][2];
870   int i, j, k, l;
871
872   if(SubFaceCubeBoundsReversed(bound)) return FALSE;
873   if(SubFaceCubeBeyondSpaceBounds(bound, space)) return FALSE;
874   if(StartOrFinishPointInsideSubFaceCube(bound, space)) return
          FALSE;
875   if(AnyFaceInsideSubFaceCube(bound, space)) return FALSE;
876
877   for(i=XAxis; i<=ZAxis; i++) {
878     for(j=0; j<=1; j++) {
879       for(k=XAxis; k<=ZAxis; k++)
880         for(l=0; l<=1; l++)
881           tempbound[k][l]=bound[k][l];
882       for(l=0; l<=1; l++)
883         tempbound[i][l] = bound[i][j];
884       tempface = newSubFace(tempbound, i, j, NONE);
885       if (!SubFaceLegalInSpace(tempface, space)) return FALSE;
886       free(tempface);
887     }
888   }
889   return TRUE;
890 }
891
892
893 void InsertSubFaceCubeIntoSpace(Num bound[3][2], Space * space
        , FaceType facetype)
894 {
895   SubFace * tempface;
896   Num tempbound[3][2];
897   int i, j, k, l;
898
899 //   printf("Attempting to insert cube ");
900 //   printBound(bound);
901 //   printf("\n");
902
903   for(i=XAxis; i<=ZAxis; i++) {
```

```
904         for(j=0; j<=1; j++) {
905           for(k=XAxis; k<=ZAxis; k++)
906             for(l=0; l<=1; l++)
907               tempbound[k][l]=bound[k][l];
908           for(l=0; l<=1; l++)
909             tempbound[i][l] = bound[i][j];
910           tempface = newSubFace(tempbound, i, j, facetype);
911           TestAndInsertSubFaceIntoSpace(tempface, space);
912         }
913       }
914     }
```

Listing 8.3: subface.c

Source code for :space.c

```
1  // space.c                                                    David
       Wagner
2  // Operations in Space
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7
8
9  #include "datatypes.h"
10 #include "space.h"
11 #include "subface.h"
12 #include "generate.h"
13 #include "segtree.h"
14
15
16
17 // Return a new Space
18
19 Space * newSpace(Num bound[3][2])
20 {
21   int i,j;
22   static Label label = 0;
23
24   Space * space = (Space *)malloc(sizeof(Space));
25
26 //   space->faces = (SubFaceSet **)malloc(6 * sizeof(SubFaceSet
       *));
27
28   space->start = space->finish = NULL;
29
30   space->label = label++;
31
32 //   space->numevents = 0;
33
34   for(i=XAxis; i<=ZAxis; i++) for(j=0; j<=1; j++){
35       space->faces[i][j]=newSubFaceSetHead();
36       space->facecount[i][j]=0;
37       space->bound[i][j]=bound[i][j];
38   }
39   for(i=XAxis; i<=ZAxis; i++) for(j=0; j<=1; j++){
40       generateSpaceDirs(space, i, j);
```

```
41          space->inside[i][j] = newSegTree2D(space->left,
42            space->right, space->bottom, space->top);
43      }
44      return space;
45
46  }
47
48  //////// These functions are moved into subface.c
49
50  // Insert a SubFace into the Space
51  //
52  //void insertSubFaceIntoSpace(SubFace * subface, Space * space
        )
53  //{
54  //   Axis axis = getFaceAxis(subface);
55  //   Sign sign = subface->sign;
56  //
57  //   if (axis == NoAxis)
58  //     prinf("Error: Inserting Subface with no axis into Space\
        n");
59  //
60  //   addSubFace(space->faces[axis][sign], subface);
61  //   space->facecount[axis][sign]++;
62  //}
63
64  // Add all six faces bounding the represented region to
65  // the space
66
67  //void insertSubFaceCubeIntoSpace(Num bound[3][2], Space *
        space)
68  //{
69  //          SubFace * tempface;
70  //          Num tempbound[3][2];
71  //          int i, j, k, l;
72  //
73  //   for(i=XAxis; i<=ZAxis; i++) {
74  //     for(j=0; j<=1; j++) {
75  //       for(k=XAxis; k<=ZAxis; k++)
76  //         for(l=0; l<=1; l++)
77  //           tempbound[k][l]=bound[k][l];
78  //       for(l=0; l<=1; l++)
79  //         tempbound[i][l] = bound[i][j];
80  //       tempface = newSubFace(tempbound, j);
81  //       insertSubFaceIntoSpace(tempface, space);
82  //     }
```

```
83 //   }
84 //}
85
86
87 // Add starting point to space
88 // return true on success
89
90 Boolean addStartPoint(Point * start, Space * space)
91 {
92         Num bound[3][2];
93
94         bound[0][0] = bound[0][1] = start->x;
95         bound[1][0] = bound[1][1] = start->y;
96         bound[2][0] = bound[2][1] = start->z;
97
98         if (SubFaceCubeLegalInSpace(bound, space)){
99                 InsertSubFaceCubeIntoSpace(bound, space, START
                    );
100     space->start = start;
101     return TRUE;
102         }
103
104   return FALSE;
105
106 }
107
108
109 // Add finishing point to space
110 // return true on success
111
112 Boolean addFinishPoint(Point * finish, Space * space)
113 {
114         Num bound[3][2];
115
116         bound[0][0] = bound[0][1] = finish->x;
117         bound[1][0] = bound[1][1] = finish->y;
118         bound[2][0] = bound[2][1] = finish->z;
119
120         if (SubFaceCubeLegalInSpace(bound, space)){
121                 InsertSubFaceCubeIntoSpace(bound, space,
                    FINISH);
122     space->finish = finish;
123     return TRUE;
124         }
125
```

```
126     return FALSE;
127 }
128
129
130 // Add a random starting point to space
131 // return true on success
132
133 Boolean addRandomStartPoint(Space * space)
134 {
135     Point * start = (Point *)malloc(sizeof(Point));
136     Num xwidth, ywidth, zwidth, xcoord, ycoord, zcoord;
137
138     xwidth = space->bound[XAxis][POS] - space->bound[XAxis][NEG]
            + 1;
139     ywidth = space->bound[YAxis][POS] - space->bound[YAxis][NEG]
            + 1;
140     zwidth = space->bound[ZAxis][POS] - space->bound[ZAxis][NEG]
            + 1;
141
142     start->x = space->bound[XAxis][NEG] + (random() % xwidth);
143     start->y = space->bound[YAxis][NEG] + (random() % ywidth);
144     start->z = space->bound[ZAxis][NEG] + (random() % zwidth);
145
146     return addStartPoint(start, space);
147 }
148
149
150 // Add a random finishing point to space
151
152 Boolean addRandomFinishPoint(Space * space)
153 {
154     Point * finish = (Point *)malloc(sizeof(Point));
155     Num xwidth, ywidth, zwidth, xcoord, ycoord, zcoord;
156
157     xwidth = space->bound[XAxis][POS] - space->bound[XAxis][NEG]
            + 1;
158     ywidth = space->bound[YAxis][POS] - space->bound[YAxis][NEG]
            + 1;
159     zwidth = space->bound[ZAxis][POS] - space->bound[ZAxis][NEG]
            + 1;
160
161     finish->x = space->bound[XAxis][NEG] + (random() % xwidth);
162     finish->y = space->bound[YAxis][NEG] + (random() % ywidth);
163     finish->z = space->bound[ZAxis][NEG] + (random() % zwidth);
164
```

```
165     return addFinishPoint(finish, space);
166   }
167
168
169   // Choose a random wall or subface in space
170   // Attempt to add a random cube attached to that wall or
          subface
171   // return TRUE on success
172
173   Boolean insertRandomSubFaceCubeIntoSpace(Space * space, char
          cubetype)
174   {
175     int count = countAllFacesInSpace(space) + 6;
176     int randomNum = random() % count;
177
178     if (cubetype == WALLCUBE || (cubetype == EITHERCUBE &&
            randomNum < 6))
179       return extendRandomWallInSpace(space);
180     else return extendRandomSubFaceInSpace(space);
181   }
182
183   // Choose a random subface in space.  Attempt to extend it
184   // randomly.  return TRUE on success
185
186   Boolean extendRandomSubFaceInSpace(Space * space)
187   {
188     SubFace * face = GetRandomFaceInSpace(space);
189     return extendSubFaceRandomlyInSpace(face, space);
190   }
191
192   // Attempt insert a cube into space attached to the indicated
          face
193   // Extend the face a random distance.  if that is not legal,
194   // try again with half the distance.  Repeat until the
          distance is 1
195   // return TRUE on success
196
197   Boolean extendSubFaceRandomlyInSpace(SubFace * face, Space *
          space)
198   {
199     Axis axis = face->axis;
200     Sign sign = face->sign;
201
202     Num width = face->bound[axis][sign] - space->bound[axis][
            sign];
```

```
203   Num randWidth;
204   Num bound[3][2];
205
206   int i,j;
207
208   if (width < 0) width = -width;
209   randWidth = random() % width;
210
211 //   printf("Width is %d\n",width);
212 //   printf("Random width is %d\n",randWidth);
213
214   for(i=XAxis; i<=ZAxis; i++) for(j=0;j<=1;j++)
215     bound[i][j]=face->bound[i][j];
216
217
218   while (randWidth > 0) {
219     if (sign == POS) {
220       bound[axis][POS] = face->bound[axis][POS]+randWidth+1;
221       bound[axis][NEG] = face->bound[axis][POS]+1;
222     } else {
223       bound[axis][NEG] = face->bound[axis][NEG]-randWidth-1;
224       bound[axis][POS] = face->bound[axis][NEG]-1;
225     }
226
227 //     printf("Attempting to insert ");
228 //     printBound(bound);
229 //     printf("\n");
230
231     if (SubFaceCubeLegalInSpace(bound, space)) {
232       InsertSubFaceCubeIntoSpace(bound, space, NONE);
233       return TRUE;
234     }
235
236     randWidth /= 2;
237   }
238
239   return FALSE;
240 }
241
242 // Choose a random wall in space.  Attempt to add a cube
243 // attached to that wall with random coordinates and of width
    1
244 // return TRUE on success
245
246 Boolean extendRandomWallInSpace(Space * space)
```

```
247 {
248   Axis axis = random() % 3;
249   Sign sign = random() % 2;
250   int i,j;
251   Num temp, randCoord[3][2], width[3];
252   Num l, r, b, t;
253
254   generateSpaceDirs(space, axis, sign);
255
256   for(i=XAxis; i<=ZAxis; i++)
257     width[i] = space->bound[i][POS] - space->bound[i][NEG] +
          1;
258
259   for(i=XAxis; i<=ZAxis; i++) for(j=0; j<=1; j++)
260     randCoord[i][j] = space->bound[i][NEG] + (random() % width
          [i]);
261
262   for(i=XAxis; i<=ZAxis; i++)
263     if (randCoord[i][NEG] > randCoord[i][POS]){
264       temp = randCoord[i][NEG];
265       randCoord[i][NEG] = randCoord[i][POS];
266       randCoord[i][POS] = temp;
267     }
268
269   randCoord[axis][sign] = space->bound[axis][sign];
270   randCoord[axis][1-sign] = space->bound[axis][sign] + 2 - (2*
        sign);
271
272 //   printf("Attempting to insert ");
273 //   printBound(randCoord);
274 //   printf("\n");
275
276   l = randCoord[generateLRAxis(axis)][generateLeftSign(sign)];
277   r = randCoord[generateLRAxis(axis)][generateRightSign(sign)
        ];
278   b = randCoord[generateTBAxis(axis)][generateBottomSign(sign)
        ];
279   t = randCoord[generateTBAxis(axis)][generateTopSign(sign)];
280
281   if (QueryRect(space->inside[axis][sign], l, r, b, t)) return
        FALSE;
282
283   if (SubFaceCubeLegalInSpace(randCoord, space)) {
284     InsertRect(space->inside[axis][sign], l, r, b, t);
285     InsertSubFaceCubeIntoSpace(randCoord, space, NONE);
```

```
286       return TRUE;
287     }
288
289     return FALSE;
290   }
291
292
293   // Determine the best axis on which to split the space.
294   // This should be the axis with the most number of subfaces.
295
296   Axis getSplitAxis(Space * space)
297   {
298     int XCount, YCount, ZCount;
299
300     XCount = space->facecount[XAxis][POS] + space->facecount[
          XAxis][NEG];
301     YCount = space->facecount[YAxis][POS] + space->facecount[
          YAxis][NEG];
302     ZCount = space->facecount[ZAxis][POS] + space->facecount[
          ZAxis][NEG];
303
304     if (XCount >= YCount && XCount >= ZCount) return XAxis;
305     if (YCount >= ZCount) return YAxis;
306     return ZAxis;
307   }
308
309
310
311   Sign getSplitSign(Space * space, Axis axis)
312   {
313     return (space->facecount[axis][POS] >= space->facecount[axis
          ][NEG] ?
314       POS : NEG );
315   }
316
317   int countAllFacesInSpace(Space * space)
318   {
319     int i,j, accum=0;
320
321     for (i=XAxis; i<=ZAxis; i++)
322       for(j=0; j<=1; j++){
323         space->facecount[i][j]=countFaces(space->faces[i][j]);
324         accum+=space->facecount[i][j];
325       }
326     return accum;
```

```
327 }
```

Listing 8.4: space.c

Source code for :block.c

```
1  // block.c                                                    David
       Wagner
2  // Functions dealing with blocks
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7
8
9  #include "datatypes.h"
10 #include "block.h"
11
12 //    Coordinate system with dimension and sign
13 //
14 //        Y
15 //        |    Z
16 //        |   /
17 //        |  /
18 //        |/ ------ X
19
20
21 // Return a New Block
22
23 Block * newBlock(Num bound[3][2], BlockType blocktype, int
       benddist)
24 {
25   Block * myBlock=(Block *)malloc(sizeof(Block));
26
27   int i,j;
28
29   myBlock->minbenddist=benddist;
30   myBlock->blocktype=blocktype;
31
32   for(i=XAxis; i<=ZAxis; i++)
33     for(j=0;j<=1;j++){
34       myBlock->bound[i][j]=bound[i][j];
35       myBlock->faceNeighbors[i][j]=NULL;
36       myBlock->blockNeighbors[i][j]=NULL;
37       myBlock->lastsweep[i][j]=-1;
38     }
39
40   return myBlock;
```

```
41
42  }
43
44  // Returns a new BlockSet node, specially marked to be
45  // the head of the BlockSet
46
47  BlockSet * newBlockSetHead()
48  {
49    BlockSet * newBlockSet = (BlockSet *) malloc(sizeof(BlockSet
        ));
50
51    newBlockSet->block = NULL;
52    newBlockSet->next = newBlockSet;
53    newBlockSet->prev = newBlockSet;
54    newBlockSet->startblock = NULL;
55    newBlockSet->finishblock = NULL;
56    newBlockSet->count = 0;
57
58    return newBlockSet;
59  }
60
61  Boolean isBlockSetHead(BlockSet * blockset)
62  {
63    return(blockset->block == NULL);
64  }
65
66  // Set the minbenddist of the block if it is not set already
67  // or if the new minbenddist is less than the old one
68
69  void updateMinBendDist(Block * block, int newminbenddist)
70  {
71    if (block->minbenddist == NOPATH ||
72      block->minbenddist > newminbenddist)
73      block->minbenddist = newminbenddist;
74  }
75
76  // Add a Block to the BlockSet
77
78  void insertBlock(BlockSet * head, Block * block)
79  {
80    BlockSet * newBlockSet = (BlockSet *) malloc(sizeof(BlockSet
        ));
81
82    newBlockSet->block = block;
83    newBlockSet->next = head->next;
```

```
84    newBlockSet->prev = head;
85    newBlockSet->startblock = newBlockSet->finishblock = NULL;
86    head->next->prev = newBlockSet;
87    head->next = newBlockSet;
88    head->count++;
89
90    if (block->blocktype == START) head->startblock = block;
91    if (block->blocktype == FINISH) head->finishblock = block;
92
93  }
94
95  // Remove the indicated Block from the BlockSet, return the
        removed block
96
97  Block * removeBlock(BlockSet * head, BlockSet * block)
98  {
99    Block * result;
100
101   block->prev->next = block->next;
102   block->next->prev = block->prev;
103
104   result = block->block;
105
106   head->count--;
107
108   if (result->blocktype == START) head->startblock = NULL;
109   if (result->blocktype == FINISH) head->finishblock = NULL;
110
111   free(block);
112
113   return result;
114 }
115
116 // Destructively union together two BlockSets
117 // Return the new head
118
119 BlockSet * unionBlockSets(BlockSet * set1, BlockSet * set2)
120 {
121
122 //   printf("Unioning Sets:\n");
123 //   printBlockSet(set1);
124 //   printBlockSet(set2);
125
126         set2->prev->next = set1;
127         set1->prev->next = set2->next;
```

```
128
129         set2->next->prev = set1->prev;
130   set2->next = set2;
131         set2->prev->next->prev = set2->prev;
132
133   set1->count += set2->count;
134
135   if(set1->startblock && set2->startblock)
136     genError(WARNING, "unionBlockSets",
137       "Unioning block sets with two start blocks");
138   if(set1->finishblock && set2->finishblock)
139     genError(WARNING, "unionBlockSets",
140       "Unioning block sets with two finish blocks");
141
142   if(set2->startblock) set1->startblock = set2->startblock;
143   if(set2->finishblock) set1->finishblock = set2->finishblock;
144
145         free(set2);
146
147 //   printf("Finishing Union\n");
148 //   printBlockSet(set1);
149
150         return(set1);
151 }
152
153
154 void addFaceNeighbor(Block * block, SubFace * faceNeighbor,
      Axis axis, Sign sign)
155 {
156   insertSubFace(block->faceNeighbors[axis][sign], faceNeighbor
        );
157 }
158
159
160 void addBlockNeighbor(Block * block, Block * blockNeighbor,
      Axis axis, Sign sign)
161 {
162   insertBlock(block->blockNeighbors[axis][sign], blockNeighbor
        );
163 }
164
165
166 Boolean coordInsideBlock(Num coord, Axis axis, Block * block)
167 {
168   checkBoundsOnAxis(block->bound, axis);
```

```
169     return (block->bound[axis][NEG] <= coord &&
170       coord <= block->bound[axis][POS]);
171   }
172
173
174   Boolean topBottomInsideBlock(Num top, Num bottom, Block *
          block)
175   {
176     if (top <= bottom && block->top <= block->bottom)
177       return(block->top < top && bottom < block->bottom);
178
179     if (top >= bottom && block->top >= block->bottom)
180       return(block->top > top && bottom > block->bottom);
181
182     printf("Error: face top and bottom orientation does not
            match block\n");
183   }
184
185
186   Boolean leftRightInsideBlock(Num left, Num right, Block *
          block)
187   {
188     if (left <= right && block->left <= block->right)
189       return(block->left < left && right < block->right);
190
191     if (left >= right && block->left >= block->right)
192       return(block->left > left && right > block->right);
193
194     printf("Error: face left and right orientation does not
            match block\n");
195   }
196
197
198   Boolean topBottomOverlapsBlock(Num top, Num bottom, Block *
          block)
199   {
200     if (top <= bottom && block->top <= block->bottom)
201       return(block->top < bottom && top < block->bottom);
202
203     if (top >= bottom && block->top >= block->bottom)
204       return(block->top > bottom && top > block->bottom);
205
206     printf("Error: face top and bottom orientation does not
            match block\n");
207   }
```

```
208
209
210 Boolean leftRightOverlapsBlock(Num left, Num right, Block *
        block)
211 {
212   if (left <= right && block->left <= block->right)
213     return(block->left < right && left < block->right);
214
215   if (left >= right && block->left >= block->right)
216     return(block->left > right && left > block->right);
217
218   printf("Error: face left and right orientation does not
        match block\n");
219 }
220
221
222 Boolean subFaceInsideBlock(SubFace * subface, Block * block)
223 {
224   generateBlockDirs(block, subface->axis, subface->sign);
225   generateFaceDirs(block, subface->axis, subface->sign);
226
227   if (!coordInsideBlock(subface->coord, subface->axis, block))
228     return FALSE;
229
230   if (!topBottomInsideBlock(subface->top, subface->bottom,
        block))
231     return FALSE;
232
233   if (!leftRightInsideBlock(subface->left, subface->right,
        block))
234     return FALSE;
235
236   return TRUE;
237 }
238
239
240 Boolean subFaceIntersectsBlock(SubFace * subface, Block *
        block)
241 {
242   generateBlockDirs(block, subface->axis, subface->sign);
243   generateFaceDirs(block, subface->axis, subface->sign);
244
245   if (!coordInsideBlock(subface->coord, subface->axis, block))
246     return FALSE;
247
```

```
248     if (!topBottomOverlapsBlock(subface->top, subface->bottom,
            block))
249       return FALSE;
250
251     if (!leftRightOverlapsBlock(subface->left, subface->right,
            block))
252       return FALSE;
253
254     return TRUE;
255   }
256
257   //
258
259   Boolean planeIntersectsBlock(Block * block, Axis axis, Num
        coord)
260   {
261           if (block->bound[axis][NEG] > block->bound[axis][POS])
262                   printf("Error: Block NEG greater than POS\n");
263           return (block->bound[axis][NEG] <= coord &&
264                   coord <= block->bound[axis][POS]);
265   }
266
267   // Split the block along the given plane Return the
268   // two resulting blocks as a BlockSet
269
270   Block * splitBlock(Block * block, Axis axis, Num coord, Sign
        side)
271   {
272           Block * myBlock;
273           BlockSet * result = NULL;
274           Num newBound [3][2];
275           int i,j;
276
277           if(!planeIntersectsBlock(block, axis, coord))
278                   printf("Error: splitting block on non-
                        intersecting plane\n");
279           if(!planeIntersectsBlock(block, axis, coord+1))
280                   printf("Error: splitting block on non-
                        intersecting plane\n");
281
282           for(i=XAxis;i<=ZAxis;i++)
283                   for(j=0; j<=1; j++)
284                           newBound[i][j]=block->bound[i][j];
285
286     if (side == POS) newBound[axis][NEG]=coord;
```

```
287     if (side == NEG) newBound[axis][POS]=coord+1;
288
289     myBlock = newBlock(newBound, block->blocktype, block->
            minbenddist);
290
291             return myBlock;
292 }
293
294
295 void labelBlocks(BlockSet * head)
296 {
297     int i=0;
298     BlockSet * currblock;
299
300     for(currblock=head->next; currblock!=head; currblock =
            currblock->next){
301       currblock->block->label=i;
302       i++;
303     }
304 }
```

Listing 8.5: block.c

Source code for :generate.c

```
1  // generate.c                                          David Wagner
2  // Generate outgoing paths from a block
3
4  #include "include.h"
5
6  #include "generate.h"
7  #include "segtree.h"
8  #include "queue.h"
9  #include "bsp.h"
10 #include "sweepplane.h"
11
12 int eventnum=0;
13 int removedevents=0;
14
15 //
16 //    Coordinate system with dimension and sign
17 //
18 //        Y
19 //        |    Z
20 //        |   /
21 //        |  /
22 //        |/_____ X
23 //
24
25
26 // Given a face F, return the axis it is perpendicular to
27 // without relying on F->axis
28
29 Axis getFaceAxis(SubFace * F)
30 {
31   Axis axis;
32
33   for (axis=XAxis; axis <= ZAxis; axis++) {
34     if (F->bound[axis][POS] == F->bound[axis][POS])
35       return axis;
36   }
37
38   return NoAxis;
39 }
40
41
```

```
42 // Which way is top, bottom, left, right front, back when
       looking
43 // at the object along the given axis in the direction of sign
       ?
44
45 Axis generateTBAxis(Axis axis) { return (axis+2)%3; }
46 Axis generateLRAxis(Axis axis) { return (axis+1)%3; }
47 Axis generateFBAxis(Axis axis) { return axis; }
48
49 Sign generateTopSign(Sign sign) { return POS; }
50 Sign generateBottomSign(Sign sign) { return NEG; }
51 //Sign generateLeftSign(Sign sign) { return !sign; }
52 Sign generateLeftSign(Sign sign) { return NEG; }
53 //Sign generateRightSign(Sign sign) { return sign; }
54 Sign generateRightSign(Sign sign) { return POS; }
55 Sign generateFrontSign(Sign sign) { return !sign; }
56 Sign generateBackSign(Sign sign) { return sign; }
57
58 // Case analysis of the above statements
59
60 //   if (axis == XAxis){
61 //     B->top = B->bound[ZAxis][POS];
62 //     B->bottom = B->bound[ZAxis][NEG];
63 //     B->left = B->bound[YAxis][!sign];
64 //     B->right = B->bound[YAxis][sign];
65 //     B->front = B->bound[XAxis][!sign];
66 //     B->back = B->bound[XAxis][sign];
67 //   }
68 //
69 //   if (axis == YAxis){
70 //     B->top = B->bound[XAxis][POS];
71 //     B->bottom = B->bound[XAxis][NEG];
72 //     B->left = B->bound[ZAxis][!sign];
73 //     B->right = B->bound[ZAxis][sign];
74 //     B->front = B->bound[YAxis][!sign];
75 //     B->back = B->bound[YAxis][sign];
76 //   }
77 //
78 //   if (axis == ZAxis){
79 //     B->top = B->bound[YAxis][POS];
80 //     B->bottom = B->bound[YAxis][NEG];
81 //     B->left = B->bound[XAxis][!sign];
82 //     B->right = B->bound[XAxis][sign];
83 //     B->front = B->bound[ZAxis][!sign];
84 //     B->back = B->bound[ZAxis][sign];
```

```
85 //  }
86
87 // This function assigns the values top, bottom, left, right,
       front,
88 // and back to the face when looking at it in the given
       direction.
89 // These values depend on the direction in which one looks
90 // at the face, but there should only be one axis along which
91 // the face is considered during sweep operations
92 // It may be considered in other directions for printing
93
94 void generateFaceDirs(SubFace * F, Axis axis, Sign sign)
95 {
96   Axis TBAxis = generateTBAxis(axis);
97   Axis LRAxis = generateLRAxis(axis);
98   Axis FBAxis = generateFBAxis(axis);
99   Sign TopSign = generateTopSign(sign);
100  Sign BottomSign = generateBottomSign(sign);
101  Sign LeftSign = generateLeftSign(sign);
102  Sign RightSign = generateRightSign(sign);
103  Sign FrontSign = generateFrontSign(sign);
104  Sign BackSign = generateBackSign(sign);
105
106  F->top = F->bound[TBAxis][TopSign];
107  F->bottom = F->bound[TBAxis][BottomSign];
108  F->left = F->bound[LRAxis][LeftSign];
109  F->right = F->bound[LRAxis][RightSign];
110  F->front = F->bound[FBAxis][FrontSign];
111  F->back = F->bound[FBAxis][BackSign];
112 }
113
114 // This function assigns the values top, bottom, left, right,
       front,
115 // and back to the block when looking at it in the given
       direction.
116 // These values depend on the direction in which one looks
117 // at the block.
118
119 void generateBlockDirs(Block * B, Axis axis, Sign sign)
120 {
121   Axis TBAxis = generateTBAxis(axis);
122   Axis LRAxis = generateLRAxis(axis);
123   Axis FBAxis = generateFBAxis(axis);
124   Sign TopSign = generateTopSign(sign);
125   Sign BottomSign = generateBottomSign(sign);
```

```
126     Sign LeftSign = generateLeftSign(sign);
127     Sign RightSign = generateRightSign(sign);
128     Sign FrontSign = generateFrontSign(sign);
129     Sign BackSign = generateBackSign(sign);
130 
131     B->top = B->bound[TBAxis][TopSign];
132     B->bottom = B->bound[TBAxis][BottomSign];
133     B->left = B->bound[LRAxis][LeftSign];
134     B->right = B->bound[LRAxis][RightSign];
135     B->front = B->bound[FBAxis][FrontSign];
136     B->back = B->bound[FBAxis][BackSign];
137 }
138 
139 
140 void generateSpaceDirs(Space * S, Axis axis, Sign sign)
141 {
142     Axis TBAxis = generateTBAxis(axis);
143     Axis LRAxis = generateLRAxis(axis);
144     Axis FBAxis = generateFBAxis(axis);
145     Sign TopSign = generateTopSign(sign);
146     Sign BottomSign = generateBottomSign(sign);
147     Sign LeftSign = generateLeftSign(sign);
148     Sign RightSign = generateRightSign(sign);
149     Sign FrontSign = generateFrontSign(sign);
150     Sign BackSign = generateBackSign(sign);
151 
152     S->top = S->bound[TBAxis][TopSign];
153     S->bottom = S->bound[TBAxis][BottomSign];
154     S->left = S->bound[LRAxis][LeftSign];
155     S->right = S->bound[LRAxis][RightSign];
156     S->front = S->bound[FBAxis][FrontSign];
157     S->back = S->bound[FBAxis][BackSign];
158 }
159 
160 Event * newEvent()
161 {
162     return (Event *) malloc(sizeof(Event));
163 }
164 
165 void fillEventData(Event * event, Axis axis, Sign sign,
        EventType eventtype,
166         SegTreeNode * data, Block * block, Axis stripeAxis,
167         Num left, Num right, Num bottom, Num top, Num coord,
168         Label label)
169 {
```

```
170     event->axis = axis;
171     event->sign = sign;
172     event->eventtype = eventtype;
173     event->data = data;
174     event->block = block;
175     event->stripeAxis = stripeAxis;
176     event->left = left;
177     event->right = right;
178     event->bottom = bottom;
179     event->top = top;
180     event->coord = coord;
181     event->label = label;
182 }
183 
184 // Generate all outgoing paths from block B, given entryData,
        the paths
185 // entering the block, and put those paths into Q, the
        priority queue.
186 
187 void processEmptyBlock(PQueue * Q, Space * S, Block * B,
        SweepPlane * plane)
188 {
189     generateBlockDirs(B, plane->axis, plane->sign);
190 
191     if (!QueryRect(plane->data, B->left, B->right, B->bottom, B
          ->top)) {
192       if (PRINT_PATH_FOUND)
193         printf("No path found to block\n");
194       return;
195     }
196 
197     if (PRINT_PATH_FOUND)
198       printf("\nPath found, Updating block benddist to %d\n",
            plane->benddist);
199 
200     updateMinBendDist(B, plane->benddist);
201 
202     if (PRINT_GENERATING_PATHS)
203       printf("\n======Generating Class A Paths=======\n\n");
204     generateClassAPaths(Q, S, B, plane);
205     if (PRINT_GENERATING_PATHS)
206       printf("\n======Generating Class B Paths=======\n\n");
207     generateClassBPaths(Q, S, B, plane);
208     if (PRINT_GENERATING_PATHS)
209       printf("\n======Generating Class C Paths=======\n\n");
```

```
210     generateClassCPaths(Q, S, B, plane);
211 }
212
213 void generateClassAPaths(PQueue * Q, Space * S, Block * B,
        SweepPlane * plane)
214 {
215   Axis outAxis;
216   Sign outSign;
217
218   for(outAxis=XAxis; outAxis<=ZAxis; outAxis++)
219     for (outSign=0; outSign<=1; outSign++)
220     generateClassAPathsFromFace(Q, S, B, plane, outAxis,
            outSign);
221 }
222
223 void generateClassAPathsFromFace(PQueue * Q, Space * S, Block
        * B,
224 SweepPlane * plane, Axis outAxis, Sign outSign)
225 {
226   Event * myEvent = newEvent();
227   SegTreeNode * mySegTree;
228   Num coord;
229
230   // Rotate space and the block into the direction of the
        sweep
231
232   generateBlockDirs(B, outAxis, outSign);
233   generateSpaceDirs(S, outAxis, outSign);
234
235   // Create a new segment tree to hold the projection
236   // With class A paths we are projecting the entire
237   // width of the block, so we just insert that range
238
239   mySegTree = newSegTree(S->left, S->right);
240   InsertRange(mySegTree, B->left, B->right);
241
242   if (plane->sign == outSign) coord = B->back;
243   else coord = B->front;
244
245   // Create the event
246
247   fillEventData(myEvent, outAxis, outSign, OutgoingPath,
          mySegTree, B,
248     YAxis, B->left, B->right, B->bottom, B->top, coord,
249     eventnum++);
```

```
250
251   // Set the benddist depending on if we are generating
252   // two or three bend class A paths
253
254   if (plane->axis == outAxis) myEvent->benddist = plane->
          benddist+3;
255   else myEvent->benddist = plane->benddist + 2;
256
257
258   insertPQueue(Q, myEvent);
259 }
260
261
262
263 void generateClassBPaths(PQueue * Q, Space * S, Block * B,
       SweepPlane * plane)
264 {
265   Sign outSign;
266
267   for (outSign=0; outSign<=1; outSign++){
268   generateClassBPathsFromFace(Q, S, B, plane, plane->axis,
          outSign,
269                 XAxis, XAxis);
270   generateClassBPathsFromFace(Q, S, B, plane, plane->axis,
          outSign,
271                 YAxis, YAxis);
272   generateClassBPathsFromFace(Q, S, B, plane, (plane->axis+1)
          %3, outSign,
273                 XAxis, YAxis);
274   generateClassBPathsFromFace(Q, S, B, plane, (plane->axis+2)
          %3, outSign,
275                 YAxis, XAxis);
276   }
277 }
278
279 void generateClassBPathsFromFace(PQueue * Q, Space * S, Block
       * B,
280 SweepPlane * plane, Axis outAxis, Sign outSign, Axis projAxis,
       Axis stripeAxis)
281 {
282   Event * myEvent = newEvent();
283   SegTreeNode * mySegTree;
284   Num coord;
285
286   // Rotate the block into the direction of the sweep
```

```
287
288     generateBlockDirs(B, plane->axis, plane->sign);
289
290     // Create a new segment tree to hold the projection
291
292     mySegTree=ProjectRect(plane->data, B->left, B->right,
293           B->bottom, B->top, projAxis);
294
295     // Rotate the block into the direction which paths are
            leaving
296
297     generateBlockDirs(B, outAxis, outSign);
298
299     if (plane->sign == outSign) coord = B->back;
300     else coord = B->front;
301
302     fillEventData(myEvent, outAxis, outSign, OutgoingPath,
            mySegTree,
303       B, stripeAxis, B->left, B->right, B->bottom, B->top, coord
              ,
304       eventnum++);
305
306     if (plane->axis == outAxis) myEvent->benddist = plane->
            benddist+2;
307     else myEvent->benddist = plane->benddist+1;
308
309
310     insertPQueue(Q, myEvent);
311 }
312
313
314 void generateClassCPaths(PQueue * Q, Space * S, Block * B,
        SweepPlane * plane)
315 {
316     Event * myEvent;
317     BlockSet * currblock;
318     SubFaceSet * currface;
319
320
321     int benddist = plane->benddist;
322     Axis axis = plane->axis;
323     Sign sign = plane->sign;
324
325     for(currblock = B->blockNeighbors[axis][sign]->next;
326       currblock != B->blockNeighbors[axis][sign];
```

```
327       currblock = currblock->next) {
328       if (currblock->block)
329       if (currblock->block->blocktype == OUTSIDE ||
330           currblock->block->blocktype == FINISH ||
331           currblock->block->blocktype == START)
332       if (currblock->block->lastsweep[axis][sign] < benddist){
333         currblock->block->lastsweep[axis][sign] = benddist;
334         myEvent = newEvent();
335         myEvent->benddist = benddist;
336         generateBlockDirs(currblock->block, axis, sign);
337         fillEventData(myEvent, axis, sign, EmptyBlock, NULL,
338           currblock->block, NoAxis,
339           B->left, B->right, B->bottom, B->top,
340           currblock->block->front, eventnum++);
341         insertPQueue(Q, myEvent);
342       }
343     }
344
345     for(currface = B->faceNeighbors[axis][sign]->next;
346       currface != B->faceNeighbors[axis][sign];
347       currface = currface->next) {
348       if (currface->face->facetype == NONE)
349       if (currface->face->lastsweep[axis][sign] < benddist){
350         currface->face->lastsweep[axis][sign] = benddist;
351         myEvent = newEvent();
352         myEvent->benddist = benddist;
353         generateFaceDirs(currface->face, axis, sign);
354         fillEventData(myEvent, axis, sign, ObstacleFace,
355           NULL, NULL, NoAxis, currface->face->left,
356           currface->face->right, currface->face->bottom,
357           currface->face->top, currface->face->front,
358           eventnum++);
359         insertPQueue(Q, myEvent);
360       }
361     }
362 }
363
364
365 void sweepPlaneEvent(PQueue * Q, Space * S, SweepPlane * plane
      , Event * event)
366 {
367   int benddist = plane->benddist;
368   Axis axis = plane->axis;
369   Sign sign = plane->sign;
370
```

```
371     if (event->eventtype == EmptyBlock) {
372       if (event->block->minbenddist == NOPATH ||
373         event->block->minbenddist > benddist - 3) {
374
375 //      printf("Updating Block with %d: ", benddist);
376 //      printBlock(event->block);
377 //      updateMinBendDist(event->block, benddist);
378 //      printf("Updated Block: ");
379 //      printBlock(event->block);
380         processEmptyBlock(Q, S, event->block, plane);
381       } else {
382         if (PRINT_OLD_BLOCK){
383           printf("Block is at least 3 bends old, clearing\n");
384         }
385         ClearRect(plane->data, event->left, event->right,
386           event->bottom, event->top);
387 //        printSweepPlane(plane);
388       }
389     } else if (event->eventtype == ObstacleFace){
390       ClearRect(plane->data, event->left, event->right,
391           event->bottom, event->top);
392 //      printSweepPlane(plane);
393     }
394     else if (event->eventtype == OutgoingPath){
395       InsertStripedRect(plane->data, event->data, event->left,
396           event->right, event->bottom, event->top,
397           event->stripeAxis);
398 //      printSweepPlane(plane);
399       generateClassCPaths(Q, S, event->block, plane);
400
401     }
402 }
403
404
405 //insert the starting block into the queue
406
407 void initPQueue(PQueue * Q, Block * startBlock, Space * space)
408 {
409   int i,j;
410   SegTreeNode * mySegTree;
411   Event * event;
412
413   for(i=XAxis; i<=ZAxis; i++) for(j=0; j<=1; j++){
414     generateBlockDirs(startBlock, i, j);
415     generateSpaceDirs(space, i, j);
```

```
416
417       mySegTree = newSegTree(space->left, space->right);
418       InsertRange(mySegTree, startBlock->left, startBlock->right
              );
419
420       event = newEvent();
421       event->benddist = 0;
422       fillEventData(event, i, j, OutgoingPath,
423         mySegTree, startBlock, YAxis, startBlock->left,
424         startBlock->right, startBlock->bottom,
425         startBlock->top, startBlock->front, eventnum++);
426       insertPQueue(Q, event);
427     }
428   }
429
430
431   // Main Algorithm
432
433   int alg(Space * space)
434   {
435     BSPNode * myBSP = NULL;
436     BlockSet * myBlocks = NULL;
437     Block * startBlock, * finishBlock;
438     PQueue * Q = newPQueue();
439     Event * event;
440     SweepPlane * sweepPlane = newSweepPlane();
441
442     time_t time1, time2, time3, time4;
443
444     time1 = time(NULL);
445     myBSP = randomized_bsp(space);
446     time2 = time(NULL);
447     printf("%g seconds to run BSP\n", difftime(time2, time1));
448   //   printBSP(myBSP);
449     myBlocks = getBlockSetFromBSP(myBSP);
450     labelBlocks(myBlocks);
451
452     if (PRINT_BLOCK_SET){
453       printBlockSet(myBlocks);
454     }
455     assignAllNeighborsInBSP(myBSP);
456
457     startBlock = myBlocks->startblock;
458     finishBlock = myBlocks->finishblock;
459
```

```
460     if(!startBlock) { printf("Error: no start block\n"); return
            NOPATH; }
461     if(!finishBlock) { printf("Error: no finish block\n");
            return NOPATH; }
462
463     printf("Start block is "); printBlock(startBlock);
464     printf("Finish block is "); printBlock(finishBlock);
465
466             printf("Space has %d faces\n", countAllFacesInSpace(
                    space));
467             if (myBSP) printf("BSP has %d nodes\n", myBSP->
                    subtreenodecount);
468             if (myBlocks) printf("BlockSet has %d Blocks\n",
                    myBlocks->count);
469
470     printf("Running MBP algorithm\n");
471     time3 = time(NULL);
472
473     //insert the starting block into the queue
474
475     initPQueue(Q, startBlock, space);
476
477     // event loop
478
479     while(!emptyPQueue(Q)){
480       event = removeMinPQueue(Q);
481       while(!emptyPQueue(Q) && eventEqual(event, headPQueue(Q)))
482         event = removeMinPQueue(Q);
483 //      printBlockSetDists(myBlocks);
484 //      printTree2D(sweepPlane);
485 //      printf("Press return to continue\n");
486 //      getChar("");
487
488       if (needNewSweepPlane(sweepPlane, event)){
489         if (PRINT_BLOCK_SET){
490           printf("\n");
491           printBlockSet(myBlocks);
492         }
493         initSweepPlane(sweepPlane, event, space);
494       }
495
496       if (PRINT_REMOVE_EVENT){
497         printf("\n=*= Removing ");
498         printEvent(event);
499       }
```

```
500
501       removedevents++;
502       sweepPlaneEvent(Q, space, sweepPlane, event);
503
504       free(event);
505     }
506     time4 = time(NULL);
507     printf("%g seconds to run alg\n", difftime(time4, time3));
508
509     if (PRINT_BLOCK_SET){
510       printBlockSet(myBlocks);
511     }
512
513     if (PRINT_EVENT_COUNT){
514       printf("%d events inserted into queue\n", eventnum);
515       printf("%d events removed from queue\n", removedevents);
516     }
517
518     if (finishBlock->minbenddist == NOPATH)
519       printf("No path\n");
520     else
521       printf("Minbends Path has %d bends\n", finishBlock->
              minbenddist);
522
523     return finishBlock->minbenddist;
524 }
```

Listing 8.6: generate.c

Source code for :queue.c

```
1  // queue.c            David Wagner
2  // This handles operations on the priority queue
3
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <strings.h>
8
9
10 #include "datatypes.h"
11 #include "queue.h"
12
13 Boolean eventGreater(Event * event1, Event * event2)
14 {
15   if (event1->benddist > event2->benddist) return TRUE;
16   if (event1->benddist < event2->benddist) return FALSE;
17
18   if (event1->sign > event2->sign) return TRUE;
19   if (event1->sign < event2->sign) return FALSE;
20
21   if (event1->axis > event2->axis) return TRUE;
22   if (event1->axis < event2->axis) return FALSE;
23
24   if (event1->coord > event2->coord) return event1->sign;
25   if (event1->coord < event2->coord) return !(event1->sign);
26
27   if (event1->eventtype > event2->eventtype) return TRUE;
28   if (event1->eventtype < event2->eventtype) return FALSE;
29
30   return FALSE;
31 }
32
33 Boolean eventEqual(Event * event1, Event * event2)
34 {
35   if (event1->benddist != event2->benddist) return FALSE;
36
37   if (event1->sign != event2->sign) return FALSE;
38
39   if (event1->axis != event2->axis) return FALSE;
40
41   if (event1->coord != event2->coord) return FALSE;
42
```

```
43    if (event1->eventtype != event2->eventtype) return FALSE;
44
45    if (event1->block != event2->block) return FALSE;
46
47    return TRUE;
48  }
49
50
51  Event * copyEvent(Event * oldEvent)
52  {
53    Event * newEvent = malloc (sizeof(Event));
54
55    newEvent->benddist = oldEvent->benddist;
56    newEvent->axis = oldEvent->axis;
57    newEvent->sign = oldEvent->sign;
58    newEvent->eventtype = oldEvent->eventtype;
59    newEvent->coord = oldEvent->coord;
60  }
61
62
63  void doubleHeapSize(PQueue * Q)
64  {
65    Event ** newHeap = malloc(2 * Q->allocated * sizeof(Event*))
          ;
66    int i;
67
68    for(i=0; i<Q->length; i++) newHeap[i] = Q->heap[i];
69
70    free(Q->heap);
71
72    Q->heap = newHeap;
73    Q->allocated *= 2;
74  }
75
76  void swapEvents(PQueue * Q, int index1, int index2)
77  {
78    Event * temp;
79
80    temp = Q->heap[index1];
81    Q->heap[index1] = Q->heap[index2];
82    Q->heap[index2] = temp;
83  }
84
85  void insertPQueue(PQueue * Q, Event * event)
86  {
```

```
87    int pos;
88
89    if (PRINT_INSERT_EVENT){
90      printf("Inserting ");
91      printEvent(event);
92    }
93
94    while (Q->length >= Q->allocated) doubleHeapSize(Q);
95
96    Q->heap[Q->length] = event;
97    pos = Q->length;
98    Q->length++;
99
100   while (pos > 0 && eventGreater(Q->heap[(pos+1)/2 - 1], Q->
          heap[pos])) {
101     swapEvents(Q, (pos+1)/2 - 1, pos);
102     pos = (pos+1)/2 - 1;
103   }
104 }
105
106 Event * headPQueue(PQueue * Q)
107 {
108   return Q->heap[0];
109 }
110
111 Event * removeMinPQueue(PQueue * Q)
112 {
113   int pos;
114   Event * result = Q->heap[0];
115
116   Q->heap[0] = Q->heap[Q->length -1];
117   Q->length--;
118   pos = 0;
119
120   while((pos+1)*2 < Q->length &&
121     (eventGreater(Q->heap[pos], Q->heap[(pos+1)*2 - 1]) ||
122     eventGreater(Q->heap[pos], Q->heap[(pos+1)*2]))) {
123     if (eventGreater(Q->heap[(pos+1)*2 - 1], Q->heap[(pos+1)
          *2])){
124       swapEvents(Q, pos, (pos+1)*2);
125       pos = (pos+1)*2;
126     }
127     else {
128       swapEvents(Q, pos, (pos+1)*2 - 1);
129       pos = (pos+1)*2 - 1;
```

```
130       }
131     }
132
133     if((pos+1)*2 == Q->length &&
134       eventGreater(Q->heap[pos], Q->heap[(pos+1)*2 - 1]))
135         swapEvents(Q, pos, (pos+1)*2 - 1);
136
137     return result;
138 }
139
140 Boolean emptyPQueue(PQueue * Q)
141 {
142     return (Q->length == 0);
143 }
144
145 PQueue * newPQueue()
146 {
147     PQueue * result = malloc(sizeof(PQueue));
148
149     result->heap = malloc(sizeof(Event *));
150
151     result->length = 0;
152     result->allocated = 1;
153
154     return result;
155 }
```

Listing 8.7: queue.c

Source code for :sweepplane.c

```
1  // sweepplane.c                                          David Wagner
2  // This file handles operations on the sweep plane
3
4  #include "include.h"
5
6  #include "sweepplane.h"
7  #include "segtree.h"
8
9
10
11 SweepPlane * newSweepPlane()
12 {
13   SweepPlane * plane = (SweepPlane *) malloc (sizeof(
       SweepPlane));
14
15   plane->data = NULL;
16   plane->axis = NoAxis;
17   plane->sign = 0;
18   plane->benddist = 0;
19 }
20
21
22 void clearSweepPlane(SweepPlane * plane)
23 {
24   if (plane->data) clearSegTree2D(plane->data);
25     plane->data = NULL;
26 }
27
28 void initSweepPlane(SweepPlane * plane, Event * event, Space *
       space)
29 {
30   clearSweepPlane(plane);
31
32   if (PRINT_NEW_SWEEP_PLANE){
33     printf("\n
           **************************************************\n
           ");
34     printf("** New Sweep Plane: Axis=%d Sign=%d Benddist=%d
           ********\n",
35       event->axis, event->sign, event->benddist);
36     printf("
           **************************************************\n
```

```
        \n");
37    }
38
39
40    plane->axis=event->axis;
41    plane->sign=event->sign;
42    plane->benddist=event->benddist;
43
44    generateSpaceDirs(space, plane->axis, plane->sign);
45    plane->data = newSegTree2D(space->left, space->right,
46      space->bottom, space->top);
47
48  }
49
50  Boolean needNewSweepPlane(SweepPlane * plane, Event * event)
51  {
52          if (plane == NULL || plane->data == NULL ||
53        plane->axis != event->axis ||
54        plane->sign != event->sign ||
55        plane->benddist != event->benddist)
56      return TRUE;
57    return FALSE;
58  }
```

Listing 8.8: sweepplane.c